June 1991

UILU-ENG-91-2232
CRHC-91-22

(2)

Center for Reliable and High-Performance Computing

FAULT-TOLERANT COMPUTING: AN OVERVIEW

R. K. Iyer
J. H. Patel
W. K. Fuchs
P. Banerjee
R. Horst

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2232 CRHC-91-22	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL (If applicable) N/A
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7a. NAME OF MONITORING ORGANIZATION NASA/SRC/JSEP
	7b. ADDRESS (City, State, and ZIP Code) NASA: Hampton, Virginia 23665 SRC: Research Triangle Park, NC 27709 JSEP: Arlington, Virginia 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA/SRC/JSEP	8b. OFFICE SYMBOL (If applicable)
8c. ADDRESS (City, State, and ZIP Code) NASA: Hampton, Virginia 23665 SRC: Research Triangle Park, NC 27709 JSEP: Arlington, Virginia 22217	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA: NAG-1-613/ SRC: 90-DP-109 JSEP: N00014-90-J-1270
	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Fault Tolerant Computing: An Overview

12. PERSONAL AUTHOR(S)

R.K. Iyer, J.H. Patel, W.K. Fuchs, P. Banerjee and R. Horst

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____		38

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	fault tolerance, reliability, test generation, simulation, modeling, measurement, and commercial designs

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this report is to outline the major concepts and developments in the area of fault-tolerant computing. Both hardware and software fault tolerance issues are addressed. The topics covered include module, function and system-level fault detection methods, redundancy and reconfiguration strategies, fault models and testing and checking in computer systems. Software fault tolerance techniques such as recovery blocks, design diversity, and checkpointing and recovery are also discussed. Major issues in modeling and evaluation of fault-tolerant systems are outlined. The design of two successful commercial systems is discussed.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ OTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

Fault Tolerant Computing: An Overview

R.K. Iyer, J.H. Patel, W.K. Fuchs, P. Banerjee and R. Horst[1]

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

June 18, 1991

[1] This work was supported by NASA grant NAG-1-613 at the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), a NASA Center of Excellence, the Semiconductor Research Corporation under grant 90-DP-109, and the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under contract N00014-90-J-1270.

Abstract

The purpose of this report is to outline the major concepts and developments in the area of fault-tolerant computing. Both hardware and software fault tolerance issues are addressed. The topics covered include module, function and system-level fault detection methods, redundancy and reconfiguration strategies, valid fault models, and coding and checking in computer systems. Software fault tolerance methods such as recovery blocks, design diversity, and checkpointing and recovery are also discussed. Major issues in modeling and evaluation of fault-tolerant systems are outlined. The design of two successful commercial systems is discussed.

1. Introduction

The study of fault-tolerant computing has paralleled the development of modern computers. One of the very early contributions was due to von Neumann [1], the designer of the first stored program machine. Von Neumann's work addressed the question of synthesizing reliable computers from unreliable components and developed the ideas of redundancy and replication that are common in many computers today. Strong impetus for fault tolerance came from the space program in the early 1960's. There was the need to build systems that would survive without naintenance for extended periods of time. Manned space flights provided a further boost to fault tolerance. Reliability techniques advanced rapidly during this period. In this initial period, interest in fault tolerance largely remained the domain of the space, defense and telephone industries. With the rapid introduction of computers into all areas of science, business and the humanities, that domain of interest has broadened significantly. High reliability and availability have become critical for efficient functioning of our modern society. In this regard, the development of VLSI techniques has provided a major impetus to the advancement of fault-tolerant computing VLSI designs have made replication and redundancy both cost effective and practically feasible.

In the past twenty years, fault-tolerant computing has matured into a broad discipline encompassing many aspects of computer design. This article is intended to provide the reader with an overview of the different thrust areas which encompass hardware and software fault tolerance. Three factors drive the interest in fault tolerance: first is the need for high reliability: second is the need for high availability. (For example, AT&T's ESS switching systems have an availability requirement of less than 2 minutes of down-time per year.) Third is the direct impact of a loss in reliability on system performance (also referred to as performability).

This article is divided into six sections. Section 2 deals with the broad subject of hardware fault tolerance. Important characteristics of hardware fault-tolerance techniques such as hardware, information and time redundancy and the development of self-checking circuits are discussed. The question of software fault tolerance is discussed in Section 3. This is an important area since software failures are fast becoming the dominant failure mode in complex computer systems. The Apollo and shuttle missions aborted due to software faults. Recently, sections of the AT&T network were virtually paralyzed due to a software bug. Section 4 addresses the questions of testing

and design for testability. Basic to the design of fault-tolerant systems is the availability of defect-free parts. Efficient testing strategies are critical to determine the presence of defects and faults. Section 5 addresses the question of evaluation, an issue which is critical from both the designer and the user perspective. Methods and tools to determine the dependability of the overall system, and to make comparative evaluations are discussed. Both, analytical and measurement-based methods are outlined. The final section discusses the design of two successful commercial fault-tolerant systems.

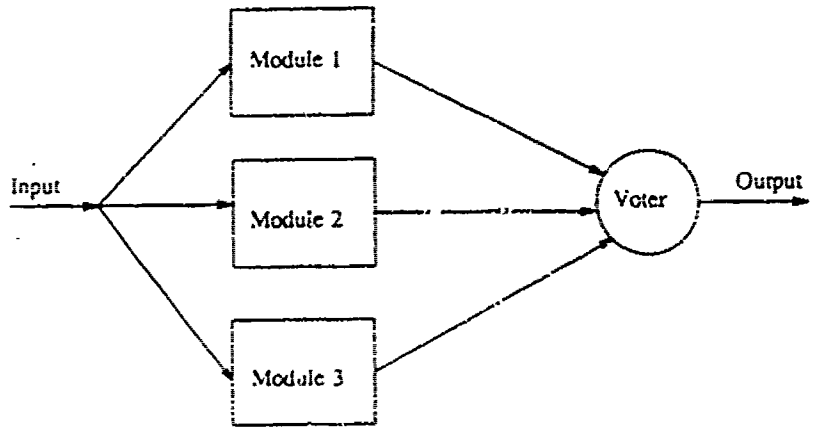
2. Hardware Fault-Tolerance Techniques

A reliable computer system needs to provide its normal level of service in the presence of hardware and software faults [2]. There are two philosophies of achieving this reliability: (1) *fault avoidance*, which is any technique to prevent the occurrence of faults in the first place; (2) *fault tolerance*, which is any technique to allow the system to behave normally despite the occurrence of faults. Fault tolerance can be implemented using one of two basic approaches: (1) *fault masking*, where the system masks the effect of a fault through some form of majority voting; (2) *fault detection and recovery*, where a system has a method for first detecting the presence of a fault, subsequently locating where the fault has occurred, next isolating the fault, reconfiguring in a spare, and restarting the system [3].

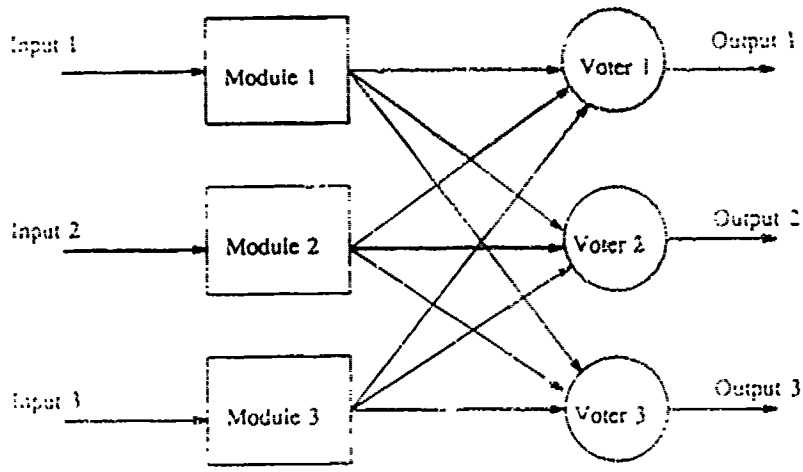
This section will describe fault-tolerance techniques for hardware faults. Hardware fault-tolerance can be achieved through the use of some form of redundancy [4]: hardware redundancy (such as spare hardware); time redundancy (such as repeating operations in time on existing hardware); information redundancy (such as some form of coding); algorithmic redundancy (modifying the algorithm running in parallel hardware with extra steps); and software redundancy (such as extra lines of software code). We will discuss the basic techniques used in each of these approaches for hardware fault-tolerance in both uniprocessor and multiprocessor systems.

2.1. Hardware Redundancy

There are three basic forms of hardware redundancy: passive, active and hybrid [4]. *Passive hardware redundancy* relies on voting mechanisms to mask the occurrence of faults by using the concept of majority voting. They do not need fault detection or system reconfiguration. The most common form of passive redundancy is called triple modular redundancy (TMR) which triplicates the hardware necessary to perform the required operations and uses a voter to determine the output of the system. In this approach, the primary difficulty is the voter. If that fails, the entire system fails. A common approach to avoid this problem is to use three voters and provide three independent outputs. Figure 1 shows the two forms of TMR. Several stages of TMR can be interconnected



(a) TMR with one voter



(b) TMR with 3 voters

FIGURE 1. Triplication and voting.

using this approach by connecting the outputs of the voters of one TMR stage via the inputs of modules of the next TMR stage. The voting can be performed by either a hardware voter (which can perform the voting very fast, but requires a lot of extra hardware logic) or a software voter (which is performed on some existing processors performing normal computations as well, but this approach is generally slow). A generalization of the TMR approach is N-modular redundancy (NMR) which uses N copies of a module instead of 3. The NASA Space Shuttle onboard computer system uses four computers on which a majority vote is performed.

Active hardware redundancy attempts to achieve fault tolerance by fault detection, fault location, and fault recovery. The most common form of fault detection is duplication and comparison which uses two identical copies of hardware, having them perform the same computations in parallel, and comparing the results as shown in Figure 2. One of the commercial products from Stratus Computers uses a pair-and-spare approach where two duplexed components are used for self-checking and fault tolerance. Two processor boards are used, where each board contains a pair of microprocessors used in duplicate and compare mode to check themselves.

Another form of fault detection includes off-line fault diagnosis, which involves applying a set of test input patterns to various components of the system and comparing the outputs to the expected outputs for each component. Other forms of fault detection include periodically interleaving normal computations with diagnostic tests, or using self-checking hardware as will be described later.

A second form of active redundancy is called standby sparing where one module is operational and one or more modules serve as standbys, or spares. Various fault detection schemes are used to determine when a module has become faulty, and fault location is used to determine exactly which module is faulty. The reconfiguration operation in standby sparing can be viewed conceptually as a switch whose output is selected from one of the modules providing inputs to the switch. Standby sparing can bring a system back into full operation after occurrence of a fault, but it requires that a momentary disruption in performance occur while reconfiguration is performed. If the disruption of processing must be minimized, hot standby sparing can be used, where the spares operate synchronously with the on-line modules, and are prepared to take over at any time. Cold standby sparing uses unpowered spares that must be powered up and initialized prior to bringing the module into active service. The advantage of cold sparing is that spares do not consume power until needed to replace a faulty module. A key

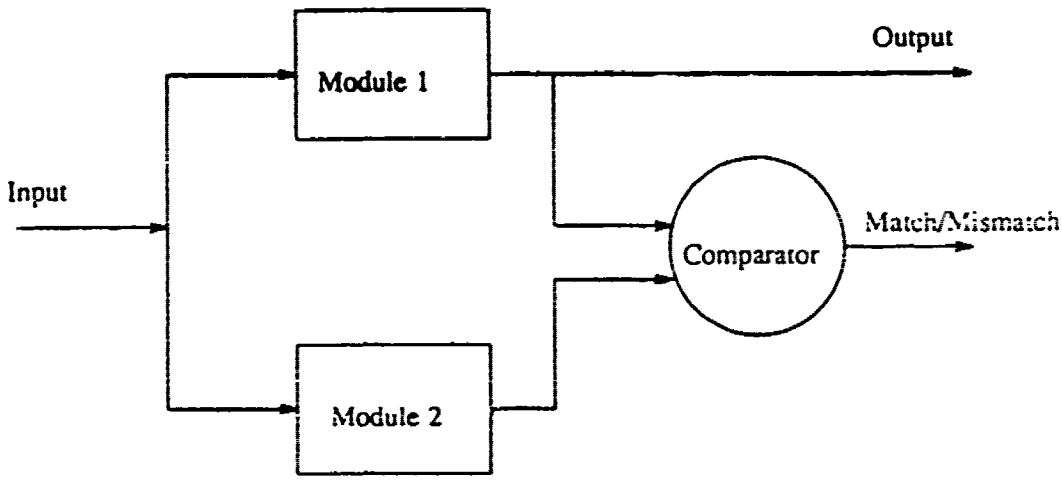


Figure 2. Duplication and comparison.

advantage of standby sparing is that in a system containing n identical modules, such as a multiprocessor, fault tolerance can be provided with $k < n$ spare modules.

Hybrid hardware redundancy combines the attractive features of both the active and passive approaches. Fault masking is used to prevent the system from producing erroneous results, and fault detection, location and recovery are used to reconfigure the system in the event of a fault. The most common form of hybrid redundancy is that of modular redundancy with spares. In this approach, a basic core of N modules is arranged in a voting configuration. In addition, spares are provided to replace faulty units in the NMR core.

2.2. Information Redundancy

Information redundancy is the addition of redundant information to data to allow fault detection, fault masking, and fault tolerance. Examples of information redundancy are error detecting and correcting codes (ECC). A code's error detection and correction properties are based on its ability to partition a set of 2^n n-bit words, each n-bits wide, into a code space of 2^m words and a noncode space of 2^n-2^m words. Each code is constructed such that a given number of errors transforms a code-space word into a word in a noncode space. Errors are detected by decoding circuits that identify any word outside the code space. Error correction is performed by more extensive decoding that uniquely associates a noncode space word with the original code word transformed by the errors.

An example of an error detecting code is the parity code, where given an n-bit word, one attaches an extra bit to convert it to an even or odd parity word. Any single bit error in the parity coded word will be detected by a simple decoding circuit using a set of XOR gates.

Within a single word, the number of errors detectable or correctable is related to the minimum separation or Hamming distance between the words of a code space, which is the minimum number of bit positions by which two words from the code space differ. Codes that need to detect d errors need to have a Hamming distance of $d+1$, and codes that need to correct c errors need a Hamming distance of $2c+1$. Error detection and correction codes vary widely in detection and correction properties, encoding and decoding complexity, and code efficiency.

The most commonly used codes are the parity check codes that are characterized by the parity check matrix, H. For example, consider a length-6 code, $n = 6$, with three information bits, $k = 3$, and three check bits, $r = 3$. The two H matrices in Figure 3 provide the same error-correcting property. Since all the columns are distinct, the code can correct all single bit errors, but the parity check circuit for H_1 is less complex than H_2. H_1 requires three 2-input XORs to compute the parity checks, whereas H_2 requires two 2-input XORs, and one 3-input XOR, hence the encoder/decoder for the H_2 code will be slower and more complex.

In high-speed memories, single-bit error-correcting and double-bit error-detecting (SEC-DED) codes are most commonly used. This is because most semiconductor RAM chips are organized for one bit of data output at a time, therefore the failure of one chip manifests itself as one-bit error. Parity codes are used routinely in

$$H_1 = \begin{array}{c} \begin{matrix} d_0 & d_1 & d_2 & c_0 & c_1 & c_2 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

$$c_0 = d_0 \oplus d_2$$
$$c_1 = d_1 \oplus d_2$$
$$c_2 = d_0 \oplus d_1$$

$$H_2 = \begin{array}{c} \begin{matrix} d_0 & d_1 & d_2 & c_0 & c_1 & c_2 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

$$c_0 = d_0 \oplus d_2$$
$$c_1 = d_0 \oplus d_1$$
$$c_2 = d_0 \oplus d_1 \oplus d_2$$

Figure 3. Parity check matrices for two simple codes.

computers to check errors in busses, memory and registers. Cyclic redundancy checks are used to detect errors in communication channels, tapes, and disks. M-out-of-N codes detect errors in control store memories. Arithmetic codes detect errors in arithmetic units like adders and multipliers. For more information about coding in reliable computer systems, the reader is referred to [6, 7].

Self-checking logic designs use the error detecting codes and some extra hardware to detect faulty logic circuits that could be single points of failures in a system. Each self-checking circuit has coded inputs and outputs. A circuit is classified as fault-secure if, for any specified fault within the circuit, the circuit never produces an incorrect output code word when stimulated by a correct input code word. A self-testing circuit outputs a noncode

word for at least one code word input for each possible fault. A totally self-checking circuit has properties of both fault-secure and self-testing circuits [8]. Self-checking circuits have been used widely in the AT&T Electronic Switching Systems 3A processors.

2.3. Time Redundancy

The basic concept of time redundancy is the repetition of computations two or more times and comparing the results to determine if a discrepancy exists. If an error is detected, the computations can be performed again to see if the disagreement remains or disappears. Such approaches are good for detecting errors due to transient faults, but cannot protect against errors resulting from permanent faults.

Another form of time redundancy to handle permanent faults modifies the way the computations are performed the second time. One approach uses alternating logic for self-dual combinational circuits [9], which performs a function on some set of inputs in one time instant, and performs the same function on the complemented input in a subsequent time step, the output of which should be the complement of the original function value of the original input. If the second value of the function is not the complement, an error is detected.

The second approach uses recomputing with shifted operands [10], which is applicable to bit-sliced organizations of hardware. In the first time step, the normal computation is performed on the operands and the results stored in a register. In the next time step, the operands are shifted left by k bits, and the output is shifted right by k bits and compared with the result of the previous computation. Any error in $k-1$ consecutive bit slices of an arithmetic or logical operation will be detected by this method. The additional hardware requirement is the three shifters, the storage register to hold the results of the first computation, and the comparator.

A variant of this method is called recomputing with swapped operands, where in the first time step, the operation is performed in the normal form. In the following time step, the upper and lower halves of the operands are swapped such that a faulty bit slice operates on opposite halves of the operands in two computations. The additional hardware requirements are in the form of several multiplexers, a storage register and a comparator.

2.4. Algorithmic Redundancy

A relatively new approach to fault tolerance is the use of algorithm-based fault tolerance which is useful in developing low-cost techniques for error detection and fault tolerance in parallel processor systems while performing specific computation-intensive applications [11]. Contrary to conventional data encoding, which is done at the word level in order to protect against errors which affect bits in a word, in algorithm-based approaches, data is encoded at a higher level. This encoding can be done by considering the set of input data to the algorithm and encoding this set. The original algorithm must then be redesigned to operate on this encoded data and to produce encoded output data. The redundancy in the encoding would enable the correct data to be recovered or, at least, to recognize that the data are erroneous. This technique has been applied to systolic arrays performing a variety of computations such as matrix operations, Fast Fourier Transform, matrix equation solvers, sorting, QR factorization, recursive least squares, filtering, and singular value decomposition [12].

We illustrate the application of an algorithm-based checking technique by an example: the multiplication of two $N \times N$ matrices. In the checksum encoding, an extra row and an extra column are appended to the original matrix, which are the sums of the elements of the columns and rows, respectively [11]. After the matrix-matrix multiplication is performed, the result matrix also preserves the checksum property. If there is an error in the result matrix element (i, j), it will be identified by verifying the equality of the sum of the row elements with the checksum for row i, and by verifying the equality of the sum of the column elements with the checksum for column j. Once the erroneous element is identified, the correct element can be reconstructed by taking the sum of all elements of that row (column) except the erroneous element and subtracting this sum from the row (column) checksum. This is illustrated in Figure 4 which shows a 5×4 row checksum encoded matrix multiplied by a 4×5 column checksum encoded matrix on a 5×5 processor array having row and column broadcasting capability to produce a 5×5 full checksum matrix.

Recently algorithm-based checking techniques have been applied on more general-purpose multiprocessors such as hypercubes [13]. Studies on actual measurements of various algorithms on a hypercube have revealed that it is possible to get very high error coverages (90-95%) for detection at relatively low cost (10-15% time overhead).

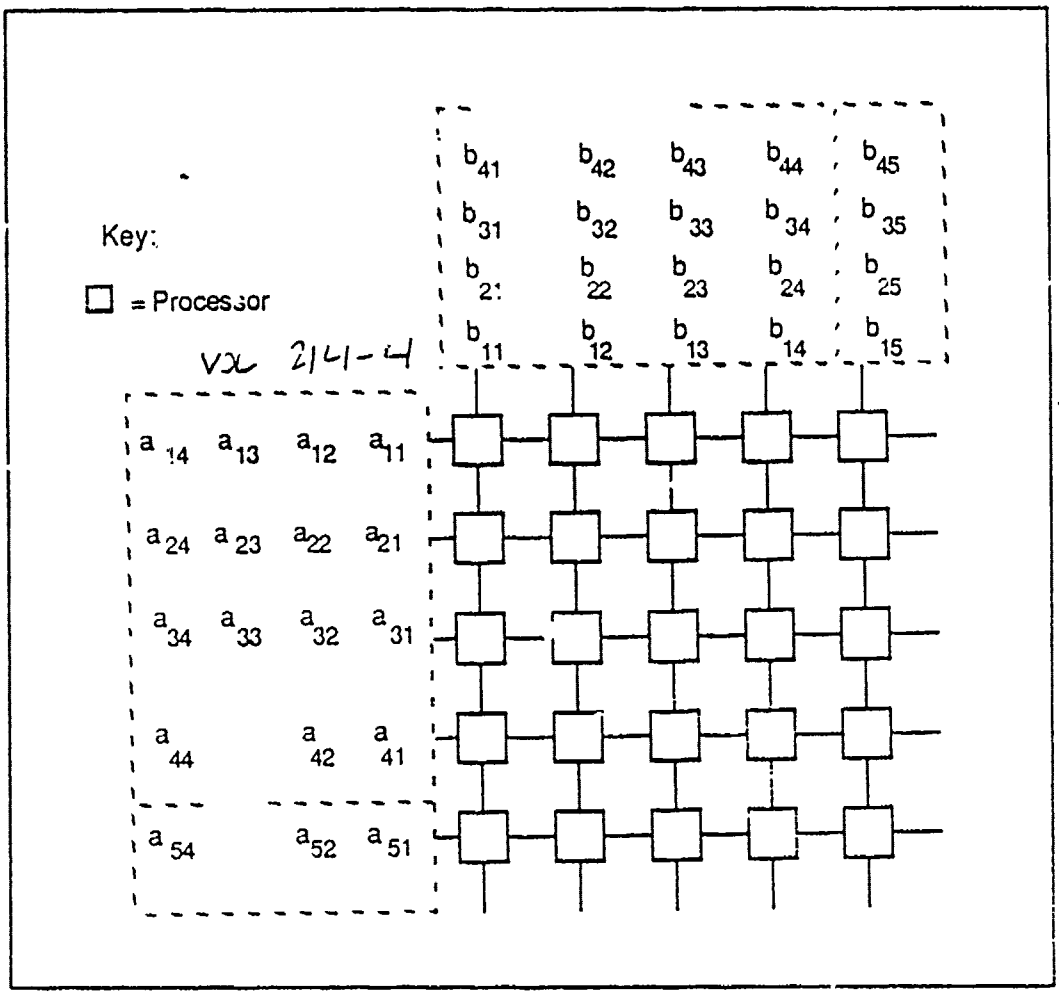


Figure 4. Algorithm-based fault tolerance for matrix multiplication.

2.5. Software Redundancy

In applications that use programmable computers, many fault-detection techniques can be implemented in software as several extra lines of code to verify the consistency of a result, such as to check the magnitude of a signal. A consistency check uses *a priori* knowledge about the characteristics of information to verify the correctness of information. In Randell, [14] such checks are application specific.

Capability checks are often performed to verify that a system possesses the capability expected. For example, if a processor has the privilege of reading or writing to a set of regions in memory under the presence of an addressing fault. Another form of capability check is used to verify if a ; can execute a specific

instruction on specific data.

3. Software Fault Tolerance

Software plays a crucial role in a computer system's ability to tolerate design, manufacturing, and wear-out faults. Faults in software are typically due to problems in design or implementation, while faults in hardware can be due to design, manufacturing, wear-out, or environmental upsets. This section presents an overview of the ways in which software design and implementation techniques can be used to detect and tolerate both software design errors and hardware faults.

The development of highly reliable software necessitates more than just software fault-tolerance techniques. The development process must include rigorous application of fault avoidance approaches, which include the correct use of formal specification languages, structured programming, formal proof of correctness, and extensive testing at all levels of implementation. Design for fault avoidance is a necessary prerequisite for effective software fault tolerance [2]. Software fault tolerance addresses the issues of detecting and recovering from residual design and implementation errors in the software and detecting and recovering from wear-out and environmentally-induced hardware faults.

3.1. Detection and Recovery from Software Faults

The fundamental approach to detecting software design errors is through exploiting diversity. Diversity in implementation and design can be in the form of acceptance tests, executable assertions, alternative software modules, or full diversity through designing and implementing multiple versions of the complete software by different teams of software engineers. Diversity can be captured through encoding knowledge of the expected behavior at various levels of the software and then comparing what is expected against what is observed. This encoding of knowledge can be at the level of the process outputs, intermediate results, system behavior, or expected algorithm behavior. The two primary approaches to software fault tolerance that provide a complete framework for capturing diversity in both design and implementation, as well as providing formal mechanisms for

error detection, error containment, and recovery, are: (1) recovery blocks [14]; and (2) N-version software [15].

3.1.1. Recovery Blocks

Recovery blocks, as developed by Randell [14], implement diversity in the form of acceptance tests and alternative software modules. Software is partitioned hierarchically into self-contained modules called "recovery blocks." Each recovery block validates its own operation and either returns correct results or notifies the system of an error. As illustrated in Figure 5 [16], each recovery block is composed of an acceptance test, the primary alternative software module, and the secondary software modules. The acceptance test is used to determine the correctness of a software module's results (error detection) and the alternative modules provide recovery from a detected error. Diversity can be captured in both the acceptance test and the secondary alternative software modules.

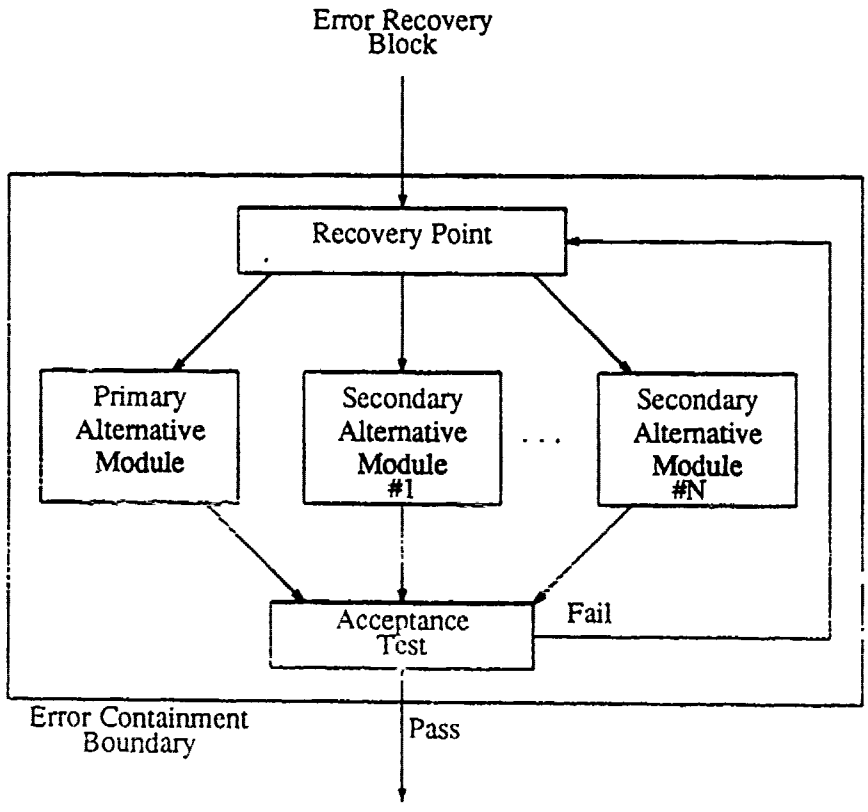


Figure 5. The recovery block approach to software fault tolerance.

An example of the acceptance test and alternative modules employed in recovery blocks can be seen in the following sorting algorithm described by Randell [14].

```
ensure sorted (S) ^ (sum(S)=sum(prior S))
        by    quickersort(S)
        else by quicksort(S)
        else by bubblesort(S)
        else error
```

The acceptance test should verify that none of the elements have changed and that the elements are, indeed, sorted. The primary algorithm can be the most efficient preferred algorithm, while the alternatives may be less efficient and are invoked only if the primary module results in an error.

Note should be made that the recovery block approach can also be implemented with distributed or parallel architectures in which the alternatives are initiated in parallel with the primary module. Also, full design diversity can be implemented if a formal specification forms the basis of each recovery block and diverse programming teams develop the alternative modules and diverse acceptance tests.

3.1.2. N Version Programming

The N version programming approach to fault tolerant software has been extensively described by Avizienis [15]. N version programming differs from recovery blocks in that it employs design diversity at the software system level through designing and implementing multiple (N) versions of the software with different teams of programmers. Instead of employing an acceptance test, N version programming utilizes voters to reach a consensus of two or more outputs among the N member versions. This approach necessarily must rely on diversity in the design to detect programming errors in the N versions of the software. If diversity is not enforced in the design and implementation, there may be an undetected or unrecoverable failure due to a single cause. Both the recovery block and N version programming approaches require a reliable execution environment for young or executing assertions and for time-efficient execution of the software modules.

3.1.3. Error Detection Techniques

Although recovery blocks and N version programming are the best known approaches that provide complete frameworks for software fault tolerance for programming errors, there are also a wide variety of individual techniques that are commonly employed outside of these frameworks. Examples of these techniques include acceptance tests and executable assertions that are commonly used to detect anomalies due to either programming errors or hardware failures [18-22]. Fail-stop tests, such as timers for detecting time-out conditions, are also common.

3.2. Software Approaches to Detection and Recovery from Hardware Faults

3.2.1. Masking and Voting

The N version programming approach is directly applicable to detection and recovery of hardware faults when the multiple versions are executed on different hardware units. This is a variant of the classic NMR (TMR) form of fault tolerance as described earlier in this article. It is possible that hardware faults can be tolerated even without diversity if the hardware and software are replicated N times and the voter is designed to be fault tolerant. However, the application of design diversity to both the N software and N hardware units can provide a line of defense against software programming faults, hardware design faults, environmental upsets, and wear-out faults [3].

3.2.2. Assertions and Alternative Execution

The recovery block approach is also directly applicable to detection and recovery from hardware failures, as well as from programming faults. As described elsewhere in this article, recent algorithm-specific techniques for encoding inputs and checking expected outputs (algorithm and behavior-based fault tolerance) have been developed for detection and recovery from hardware faults. These algorithm-specific approaches are a combination of hardware and software fault tolerance in that they employ algorithm modification for detection and recovery from hardware failures.

3.2.2.1. Fault-Tolerant Data Structures

Linked data structures provide a specific example of specialized techniques for error detection and recovery. The initial work concerning detection of errors in links (structural integrity) utilizing redundant links was developed by Taylor, Morgan and Black. Detection and correction algorithms for data structures, when used concurrently with normal data structure accesses, typically degrade performance. If data structure checking operations are performed in a small locality around currently accessed node then error detection and correction can potentially be performed concurrently with normal data structure accesses without severely degrading the system performance. In addition, an arbitrary number of errors in the data structure may be detected and corrected assuming not too many errors exist within a given locality.

One example of such a technique for detecting and correcting structural errors in data structures is the *virtual backpointer* [24]. The virtual backpointer provides the capabilities of structural error detection and correction as well as the generation of backpointer values used in backward traversals. The Virtual Double-Linked List (VDLL) is a uniform data structure that employs the virtual backpointer for local error detection and correction and for backward traversals. The VDLL requires the same storage space as the standard double-linked list (DLL), and retains the simplicity of the DLL, since it is possible to move directly from a node to its parent, using the virtual backpointer. In addition, the VDLL has enhanced error detection and correction capabilities over the DLL. An example of the VDLL is shown in Figure 6. In addition to the normal forward pointer, a virtual backpointer is stored in each node. The virtual backpointer is a function of the address of the previous (back) node and the current forward pointer. It can be shown that it is possible to detect any two errors in a VDLL and correct any single error for forward moves.

3.2.3. Reexecution Through Checkpointing and Rollback

Checkpointing is an important technique for recovery after error detection by means of rollback reexecution of a process. Checkpointing schemes can be broadly classified as full or incremental checkpointing. The former saves the entire active state space of a process while the latter saves the difference between the current and a previous state space. A checkpointing scheme can be implemented at the system or application level.

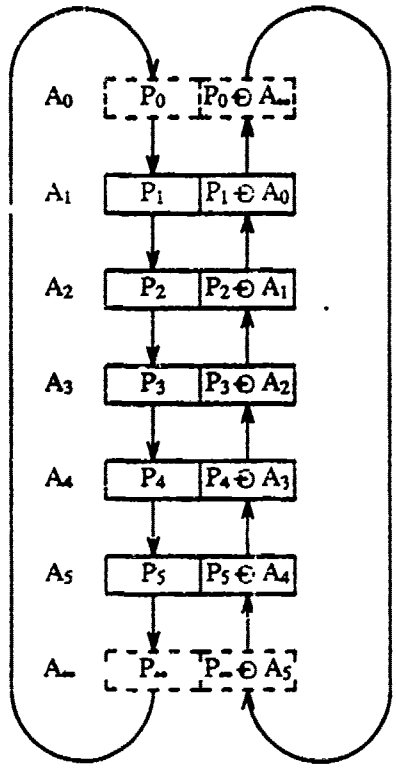


Figure 6. Example VDLL robust data structure.

Research on classic checkpointing and rollback recovery has been extensive [16-20]. Graph-theoretic methods by which the programmer can decide where to insert checkpoints have been developed. The program is decomposed by the programmer into a sequence of tasks between which the checkpoints can be inserted. It is assumed that the execution time, the checkpoint time, and the recovery time of each of these tasks is known in advance. With this information, the algorithms can determine the optimal places to insert checkpoints so that the maximum checkpoint time, the expected checkpoint time, or the expected run time is minimized.

3.2.3.1. Compiler-Assisted Checkpointing

Compiler-assisted techniques for implementing a checkpointing scheme have recently been developed [25]. This approach can achieve, in some instances, both programmer transparency and reduced checkpoint size without modification of the hardware or operating system. Compiler-generated sparse potential checkpoint code is used to maintain the desired checkpoint interval. The compiler-assisted approach to checkpointing utilizes several measurement and adaptive learning techniques to exploit periodic reduction in memory requirements to reduce the size

of checkpoints, when possible [25].

3.2.3.2. Error Detection and Recovery in Distributed Systems

Fault tolerance in distributed systems has long been the focus of extended research [26-29]. Practical applications of this research include distributed databases and real-time systems. Important elements in fault-tolerant distributed systems include reliable communication and synchronization protocols, reliable storage media and storage algorithms (replication), and reliable individual processing nodes [26].

One of the important concepts in distributed systems is that of atomic actions and commit protocols. They are used to ensure the completion or rollback of transactions. Nested transactions have been proposed as a mechanism for encapsulating the synchronization and failure properties of distributed systems [26]. Recovery in distributed database systems is often implemented through rollback of transactions and use of shadow paging or undoing a write ahead log. Network partitioning and data replication have also been used to tolerate node failures [29]. Examples of protocols developed to deal with data partitioning and replication include weighted voting, majority consensus, and quorum-based commit [26].

3.2.3.3. Recovery through Checkpointing and Rollback in Shared Memory Parallel Multiprocessors

Since different processors in a shared memory multiprocessor system can access the same memory space, a rollback of one process in multiprocessor systems may require a rollback of another, as well. It has recently been shown that through appropriate modification of cache coherence protocols, periodic checkpointing of the cache contents can be made into the shared memory in such a way that a consistent shared memory state is maintained [30]. The consistent shared memory state ensures that only the process encountering the error, resulting from a processor transient fault, is involved in the rollback recovery at the point of error detection and no rollback propagation is required. Without rollback propagation, rapid rollback recovery is thus achieved simply by invalidating the cache contents and then restarting the process from the checkpoint after reloading the program counter and registers.

In the multiprocessor cache-based checkpointing approach there are two instances in which a process has to be checkpointed. The first instance occurs whenever a cache block modified since the last checkpoint is to be written back to the shared memory, which happens when a cache block is replaced on a cache miss. The second instance occurs when another processor is to read a dirty block modified in a processor's cache since its last checkpoint. Checkpointing is initiated by the cache controller in hardware and is transparent to system or application software. Checkpointing a process includes flushing the cache blocks modified since the last checkpointing session and saving the processor internal registers.

Once a processor error is detected, all cache blocks, except those that are *unwritable* in the private cache of that processor are invalidated. The processor internal registers are reloaded and execution is restarted. Cache misses occurring when a processor resumes execution are serviced by data from the global checkpoint which is stored in the shared memory and caches of other processors. The cache coherence protocol enforces delivering the correct version of data if another cache has a block which matches the miss.

To integrate the multiprocessor cache-based checkpointing scheme into cache coherence protocols, one extra state for a cache block is introduced. A modified cache block is split into two classes: writable-modified and unwritable. Figure 7 illustrates the Illinois cache coherence protocol [30], which has been modified by adding one state to incorporate the cache-based checkpointing scheme.

There have been numerous recent developments in implementing shared memory programming environments on distributed memory multiprocessors. Typically called *distributed shared memory*, such environments utilize memory coherence protocols to implement the shared memory paradigm in software. Memory coherence-based checkpointing techniques have been developed for distributed shared memory, similar to those for cache-based checkpointing [31]. A checkpoint occurs by an individual processor if a page of memory is requested by another processor that has been modified since the last checkpoint. Rollback is implemented by simply invalidating all local pages and restoring processor registers.

The advantage of most checkpointing schemes that are embedded in the memory management protocol is that they are transparent to the application programmer. They potentially can be implemented so as to minimize performance degradation. However, their disadvantage is that it is not easy to change or control the frequency of

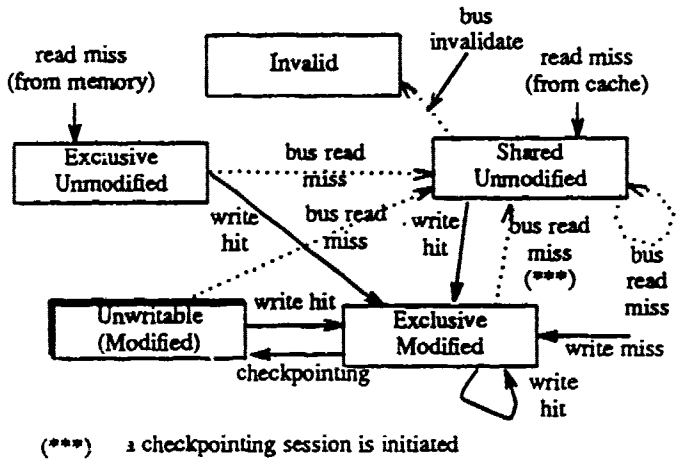


Figure 7. Cache coherence protocol incorporating cache-based checkpointing.

checkpointing with such approaches. In general, an integrated approach applying a hierarchy of checkpointing strategies (memory management, operating system, and application level) is necessary to effectively address the shortcomings of individual techniques.

4. Testing

Testing is the process of discovering faults, defects or malfunctions in the system under test. The test process consists of exercising the system with a set of *test vectors* and analyzing its response for correctness. Design of a reliable computer system involves many levels of abstractions. Typical levels of abstractions from lowest to the highest are: logic level, register level, instruction set level, processor level and system level. Testing closely follows these levels of abstraction and separate testing procedures apply to each level. The stimuli and responses defining a test experiment use the information being processed relevant to each level. For example, testing at logic level involves binary vectors or vector sequences. Similarly, higher levels of testing involve machine instructions, arithmetic numbers, textual data, messages, procedure and application programs. Thus, testing is a very broad term encompassing many different activities and environments. Testing theory and practice is most

mature at the logic level. Testing at logic level is well-defined and rigorous. Testing becomes less precise with increasing levels of abstraction. At the system level testing is largely *ad hoc* and based on intuition and experience.

Testing begins at the semiconductor chip manufacturing level and continues at higher levels of assembly and packages, such as printed circuit boards, the hardware system and finally the complete system including the operating system and application software. The initial purpose of testing at chip level is for diagnosis of chip defects. The diagnosis process identifies what is specifically wrong with a bad chip. The information is valuable in fine-tuning the semiconductor fabrication process. Defects at this level consist of open or shorted wires and transistors, slow transistors, too high a power consumption, weak drivers and so forth. Once the process is fine-tuned and mature, the main purpose of chip level testing becomes that of sorting good chips from bad chips. This is where the most rigorous testing takes place. The cost of testing a chip is a significant fraction of the overall cost of manufacturing. However, the cost of testing the same chip at higher levels is even higher. Experience shows that a defective chip, escaping as a good chip due to an imperfect testing procedure, costs 10 times as much to test on a printed circuit board. The cost includes increased difficulty in testing, and locating and replacing the bad chip on the board. If the defective chip escapes detection at the board testing, it costs 100 times as much to locate and fix at the processor testing level. The cost of a defective chip increases 10 fold at every level. Therefore, the goal of chip testing is a near perfect differentiation between a good chip and a defective chip [32].

Testing continues after a system becomes operational and deployed in the field. The purpose of testing in the field can be of a preventive nature or can be for repair of an unoperational system. Testing for preventive maintenance locates defective chips or boards which have not been exposed in the normal operation of the system. Such unexposed defects are called latent failures. Eventually they will be exposed by some system operation. In highly reliable systems, latent failures reduce the fault tolerance capabilities of the system time. if such failures are allowed to accumulate. Therefore, highly reliable systems require periodic testing to flush out latent failures. Testing for diagnosis and repair consists of narrowing down the location of a fault to a field-replaceable unit (FRU). An FRU for a computing system is typically a board or a multi-chip module. Locating the fault to the chip boundary is very difficult and expensive and therefore repair of the board is postponed until the board can

be returned to a repair facility. The testing discipline can be broadly classified into these fields: Fault Modeling, Fault Simulation, Automatic Test Pattern Generation (ATPG), Design and Synthesis of Testable Circuits, and Built-In Self-Test (BIST).

4.1. Fault Modeling

For testing purposes, all physical faults are abstracted/modeled to a level appropriate for the component under test. At the chip level the most widely accepted fault model is that of line *stuck-at-0 or stuck-at-1*. In practice, the stuck-at fault model is further restricted to *single stuck-at fault* model, meaning the test procedure assumes that there is, at most, one fault in the circuit. This assumption reduces the test generation complexity. Physical defects such as an open connection, or a short to ground or power can be modeled as constant logic 0 or 1 in many situations. Also, experience shows that even when some physical defects do not behave as stuck-at faults, the test patterns that detect all single stuck-at faults also detect a large percentage of such physical defects. Another advantage of the stuck-at fault model is that it is a logic model and therefore many of the results from Boolean algebra can be used in the test generation and analysis of such faults. For this reason, stuck-at fault is a widely accepted fault model for most digital circuits. The stuck-at fault model is useful in proving structural integrity of a circuit within the constraint of Boolean equivalence. There are also other logic fault models that have become very important for highly reliable components. A short between two wires is called a *bridging fault*, which is logically equivalent to a wired AND or OR depending on technology and relative electrical strengths of two opposing polarity signals on the short. Other physical faults, such as resistive shorts and opens, trapped charges in gate oxide of MOS transistors and weak transistors, can adversely affect the original timing specification of a circuit. Such faults are modeled as *delay faults*.

The next higher level of abstraction for fault models is broadly called *functional faults*. A functional fault is simply an incorrect execution of a function. For specific functions these incorrect behaviors can be narrowed down. For example, a restricted fault model for an address decoder might say, "for input address i the faulty decoder incorrectly selects address j." A slightly more general model might say, "the decoder selects nothing, or selects j, or selects i and j" and a very general functional fault model might say, "the decoder selects any subset

(including null) of all addresses." Similarly functional fault models have been derived for many other functions such as, adders, multipliers, arithmetic and logic units of a processor, PLAs, micro-sequencers, instruction set processors and memories. We give two more examples, one for an n-bit adder and the second for an instruction set processor.

A restricted functional fault model for an n-bit adder is that the sum differs by $\pm 2^i$. Such a model can be derived from the assumption that, at most, one internal carry or an output sum is faulty. If we extend the physical faults to a full adder in a ripple-carry adder implementation, then it can be shown that the functional fault model for the n-bit adder is that the sum differs by $(\pm 2^i \ \pm 2^{i+1})$ in the presence of a fault.

For a processor, a widely accepted functional fault model is that when instruction I has to be executed, the processor: (a) does not execute any instruction; or, (b) it executes some other instruction J; or, (c) it executes I and some other instruction J. Such abstraction allows one to generate a test set for the processor without the structural gate level information.

The higher the level of abstraction of the fault model, the more generally applicable it becomes, independent of specific implementations. At any level of abstraction the fault model can be very general or very restrictive. Very general fault models give a higher degree of confidence in the quality of the test set, in the sense that the test set will cover a large number of physical failures and a broad class of failures (e.g., stuck-at, bridging and delay). Restrictive functional fault models correspondingly give a lower quality of test sets. More information can be found in [33, 34].

4.2. Fault Simulation

Fault simulation consists of simulating a circuit in the presence of faults. The most common fault model used by fault simulators is the single stuck-at fault. However, in theory any fault model can be used during simulation. By comparing the outputs from simulation of fault-free circuits with the faulty circuit one can determine if the fault is detected by the applied test. For a given test set, T, a fault simulation produces the list of faults which are detected by T. The number of detected faults expressed as a percent of all faults is called *fault coverage*. Fault coverage is a measure of the quality of a test set. The process of finding the fault coverage of T is called

fault grading the test set T. A perfect test set will have 100% fault coverage for the assumed fault model. A component passing 100% test set may still have other faults not covered by the assumed fault model. There are several applications of fault simulation. Fault simulation is used: 1) in fault grading a given test set; 2) in diagnosis of a faulty circuit; 3) in automatic test generation, and; 4) in verification of error detection/correction circuits in highly reliable systems.

Fault grading a test set is the most common use of fault simulators. A test set derived using a higher fault model may be graded for lower fault models. For example, circuit designers use functional verification tests to check for design errors. The same tests are often used to test the circuits for stuck-at faults. Therefore, the fault simulation is used to evaluate the effectiveness of a functional test as a stuck-at test. If the fault coverage is not satisfactory, the designer can add more functional vectors to improve the fault coverage. Thus indirectly, the fault simulator is used to generate test vectors for a circuit.

In the diagnosis applications, a fault simulator is used to generate *fault dictionaries*. A fault dictionary is a list of faults detected for each test vector. Additionally, a fault dictionary may also store the actual output response for each fault, or a compressed version of the response, called a *signature* of the fault. The diagnosis process (identification and location of the fault) relies on matching the response (or signature) from the circuit under test to the simulated response (or signature) stored in the dictionary.

Fault injection experiments are an important aspect in the design and verification of highly reliable circuits. Hardware experiments are slow and expensive and very limited in the types of fault that can be injected. Fault simulators on the other hand are easy to use and are flexible in terms of fault type, location and method of injection. Fault simulators can also evaluate the effects of several thousand faults in a single pass.

The simplest method of simulating faults is the *serial fault simulation*. It consists of taking the fault-free circuit and transforming it into a faulty circuit by injecting one fault and then simulating the circuit with a standard logic simulator. The main advantage of this method is that no special fault simulator code is needed. In addition, it can simulate just about any type of fault. However, the serial method is very time-consuming, considering the fact that a 10,000 gate circuit can have close to 50,000 single stuck-at faults. This time can be reduced by appropriately simulating many faults simultaneously. Three well-known methods are: 1) Parallel Fault Simu-

lation; 2) Concurrent Fault Simulation, and; 3) Deductive Fault Simulation.

Parallel fault simulation exploits the word parallel operations of a computer by using each bit in the word to represent a different fault. Thus, one can simulate 32 to 256 faults in a single pass depending on the machine used. Another efficiency added to most practical parallel fault simulators is the use of event-driven simulation techniques. Experience shows that a fault causes only a few logic values to change from fault-free values. Therefore, an event-driven fault simulator will need to execute very few events (gate evaluations). Concurrent fault simulator is also an event-driven fault simulator. It keeps all faulty machine states but only simulates differences between a fault-free and faulty machine. Deductive fault simulation is a symbolic simulation method and it deduces faulty behavior of all faulty machines in one pass (subject to available memory). The operations used in the deductive simulator are the union and intersection of symbolic fault lists. The execution speed of the above three methods depend to a large degree on the programming techniques and as a result are hard to compare purely based on methodology. Parallel is the easiest to implement of the three. Deductive is potentially the fastest for stuck faults in synchronous sequential circuits, but implementation complexities may make it slower than parallel. The concurrent is the most general and flexible in terms of extending it to include detailed circuit timing any type of fault behavior, and higher level functional models. For further reading on fault simulation see [33, 35, 36].

4.3. Test Generation

The simplest method to test a circuit is to subject it to random test patterns. In fact, it is quite an acceptable method for many circuits. One can use the fault simulator to calculate the fault coverage and add more random vectors if the coverage is not sufficient and iterate the process until desired coverage is reached. However, to achieve a high fault coverage experience shows that many circuits require an inordinate number of random patterns. The cost of fault simulating a large number of patterns could be far more than using a non-random algorithmic method of test generation to achieve the same coverage. There are several such *deterministic test generation* methods. Test vector of a stuck-at fault in combinational logic implementing a Boolean function F can be derived by taking the Boolean function F' of the faulty circuit and forming F XOR F'. Any vector that produces F XOR F'=1 is a test vector for that fault. In practice, this procedure of taking symbolic Boolean functions of

faulty and fault-free circuits and forming exclusive-or is quite time-consuming and getting a vector that makes the F XOR F' function a 1 is a known hard problem. Efficient test generators are based on one of the two known algorithms: 1) Roth's D Algorithm and 2) Goel's PODEM algorithm. Both of these methods use a 5-valued algebra ($0, 1, D, \bar{D}$, and X). D is a symbol representing a logic 1 in fault-free and logic 0 in a faulty circuit. Similarly, symbol $\bar{D}$ represents logic 0 in fault-free and logic 1 in a faulty circuit. X is an unknown value. In both methods, the objective is to justify logic values on various lines in the circuit to accomplish a) Fault Excitation and b) Fault Propagation. Fault excitation is the process of applying a logic value opposite to the stuck-at fault value. Fault propagation is the process of applying inputs such that a fault effect (i.e., signal D or $\bar{D}$) is propagated to an output of the circuit. The D-algorithm assigns appropriate logic values locally to a fault site and then makes assignments forward or backward in the circuit to justify the assigned value. These assignments are further justified by more assignments to other lines. This process is iterated until all internal assignments are justified solely by primary input assignments. During the justification process of an assigned logic value on a line, conflicts of signal values may arise on some other lines, in which case, the assignment must be undone. This is a systematic trial and error procedure, and it will find a test vector if one exists. In PODEM, assignments are made only to primary inputs. PODEM is a branch-and-bound search method, in which the inputs are assigned one at a time and the effect of each assignment is propagated before another primary input is assigned. If the effect of an assignment causes a bounding condition then the assignment is backtracked, and reassigned a different value. In both PODEM and D-algorithm, the procedures will find a test vector if one exists or will determine that the fault can not be detected. In the worst case, both procedures must try all binary combinations of the inputs. The worst case rarely happens in real circuits, however, the procedure can sometimes make a large number of backtracks. A large number of backtracks occur mostly for faults which are not detectable. Undetectable faults are associated with redundant gates or lines in a circuit. For high reliability it is important to remove any unintentional redundancy in the circuit. There are no other algorithms which detect redundancy more efficiently than a test generation algorithm. As a result, test generation algorithms are also used in multi-level circuit minimization procedures to remove unnecessary gates.

Test generation for synchronous sequential circuits are extensions of combinational test generation algorithms. The extension is based on transforming a sequential circuit into an iterative combinational logic array (see

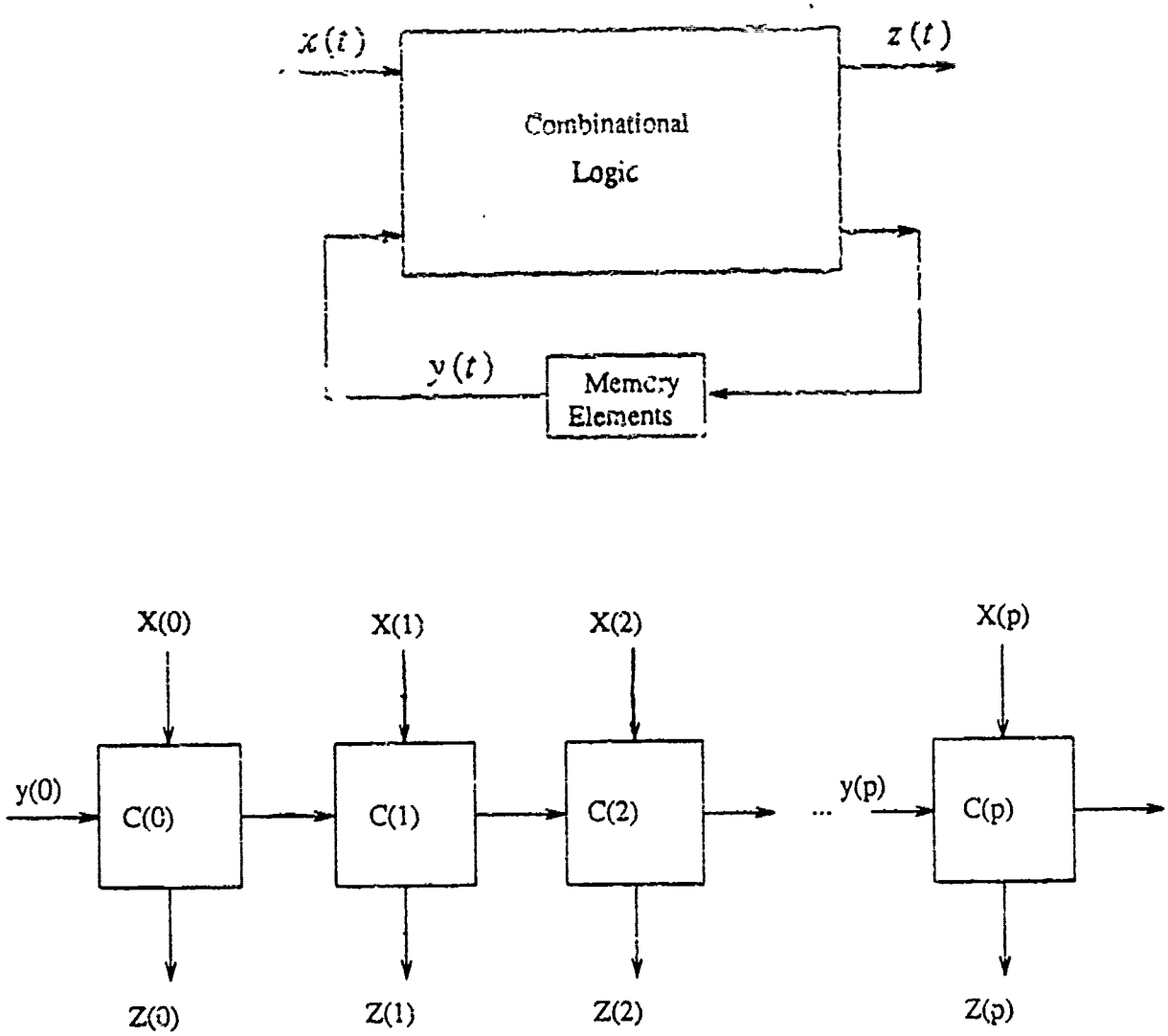


Figure 8. Time frame expansion of sequential circuits. $x(t)$, $y(t)$, $z(t)$, are signal values at time t. $C(0), C(1), C(2)$ are all identical copies of the combinational portion of the sequential circuit.

Figure 8). One cell of the array is called a *time frame*. In this transformation each flip-flop is modeled as a combinational element with input equal to current state and output equal to the next state of the flip-flop. The iterative array is simply a very deep combinational circuit. The number of cells (time frames) in the array correspond to the number of vectors in the test sequence. It can be shown that any detectable fault can be detected in 4^n vectors in a sequential circuit with n flip-flops. Therefore, the worst case bound on the number of time frames needed is 4^n, which is generally too large for any practical circuit. Therefore, during the test generation process the number of time frames are dynamically expanded, as needed. One additional complexity for sequential circuits test generation is that of initialization of flip-flops. If a reset sequence is given then it simplifies the problem somewhat. However, one has to be careful that the reset sequence can become invalid in the presence of a fault. If the reset sequence is not given, the test generator must find such a sequence. Sequential test generators can be helped by high level information such as state transition diagrams. In most cases, though, the state transition diagrams are either too large or not given, therefore the applicability of such a test generator is limited.

In addition to combinational test generator, there are *simulation-based* test generators, which is a trial-and-error approach. First, a set of trial vectors are applied and fault simulated. Based on the fault simulation results the "best" vector is retained and added to the test sequence. The process is repeated until desired fault coverage is reached. The "best" vector is defined by some cost criteria such as, number of new faults detected, or number of flip-flops set. The selection of the trial set is somewhat random but can be constrained to meet certain timing requirements. For example, we can restrict the successive vectors to differ by, at most, one bit to prevent races in asynchronous circuits. An important advantage of this method is that it is more general than combinational-based test generators. Any circuit and any fault type the simulator is able to handle is acceptable for this test generation method. One disadvantage is that, in some circuits, to achieve very high fault coverage the number of trial vectors that need to be simulated can be very large, and the resulting test sequence also tends to be very long compared to the deterministic approach described above.

Finally, there are test techniques specific to certain functions such as, adders, multipliers, iterative logic arrays, random access memories (RAM), associative memories and microprocessors. These techniques are termed *functional testing*. The term functional testing comes from the fact that each function specific test procedure

assumes a very precise functional fault model. For example, for adders and multipliers which are made of 3-bit full adders, the fault model assumed is that the fault affects the truth table of full-adder in any way. If one assumes that at most one-full adder is faulty, one can derive tests analytically which are far more compact than possible with automatic test generators. Ripple carry adders can be tested with 8 test vectors, the number 8 is constant independent of the length of the adder. Such regular structures with a constant number of tests independent of the number of cells are called *C-testable*. Ripple carry adders and two-dimensional combinational multipliers and many other iterative logic structures have proven to be C-testable.

Difficulty in memory testing is not how to generate a test set, but what realistic fault models to use and how to get a short test for such faults. The complexity of test length is very important in memories because of the number of bits involved in present RAMs. For example, if a test length grows as the square of the number of bits in the memory then a 1-Megabit RAM will require the order of 10^{12} test vectors. Most commonly used functional fault models for RAM are: bits stuck-at 0 or 1; faults in address decoder resulting in failure to address a bit; addressing the wrong bit; addressing more than one bit; coupling faults between two bits resulting in unwanted read-write operation on a coupled-bit; pattern sensitive faults resulting in failure of read or write of a bit in the presence of a specific bit pattern in the neighboring bits; and so on. In memory testing, the single fault assumption is not used. Furthermore, no upper bound on the number of faults is assumed. For all of the above functional fault models efficient test algorithms have been derived. Resources for more information on test generation methods are [33-38].

4.4. Design for Testability

In spite of major advances in test generation and fault simulation techniques, testing of digital systems still remains a very difficult problem. Testing cost remains a significant fraction of the overall cost of manufacturing VLSI chips. The complexity of test generation and cost of testing can be reduced by the process of *design for testability* (DFT). Two important factors in a testable circuit are *controllability* and *observability* of individual nodes in the circuit. Controllability is the ability to establish a specific signal value at an internal node in a circuit by setting values on (directly accessible) inputs. Observability is the ability to determine the signal value at any node

by setting values on inputs and observing subsequent outputs.

Most DFT techniques either resynthesize an existing design or add extra hardware to the design. Resynthesis systems remove most redundancies in combinational circuits. In sequential circuits, the resynthesis system encode the states to make them easier to reset, control and observe. All DFT methods affect the original design in terms of chip area, I/O pins and speed. The goal of a DFT method is to achieve the desirable testability with minimal overhead. The cost benefit of the DFT is hard to quantify in real money. Since the DFT benefits are spread over many factors such as reduction in test generation time, enhanced quality (fault coverage) and hence reduction in return rate of bad parts. It can also affect test length, test application time, tester memory, diagnosis and field maintenance time. Because of a lack of precise quantitative cost-benefit analysis, manufacturers, designers, test engineers and users disagree a great deal in their assessment of cost benefits of DFT.

A great deal of testability techniques are *ad hoc*. For example, adding reset lines, partitioning large circuits, removing redundancies, inserting control points and observation points (test points), converting asynchronous to synchronous logic, breaking long feedback paths, breaking long counters and shift registers into smaller parts, and so on. Figure 9 shows test point insertion. Addition of test points can result in too many I/O pins on a chip. The pin overhead can be reduced by employing a *scan register* to control and observe the test pins. A scan register is simply a shift register with parallel load capability. A typical scan register has 4 I/O pins: shift data-in, shift data-out, load, and shift clock. Test data is shifted in from outside and applied to the control points. The responses are captured from observation points into the scan register with a load signal and then shifted out for later analysis. The scan register trades off test point I/O pins with increased area and increased test time.

There are several methods to select test points in a circuit. Some methods are *ad hoc*, some based on analysis, and test generation and fault simulation, and some based on converting sequential circuits into combinational logic. Good candidates for *ad hoc* selection of test points are: lines with high fanouts, global feedback paths, (intentional) redundant lines, flip flops, addresses, data and control signals of embedded memories, and internal clocks. Analysis-based methods use quantifiable controllability and observability measures, or other measures such as sequential depth, and number of feedback paths passing through a node. These are used in putting test points at the least controllable and observable nodes. A global optimization program will put test points

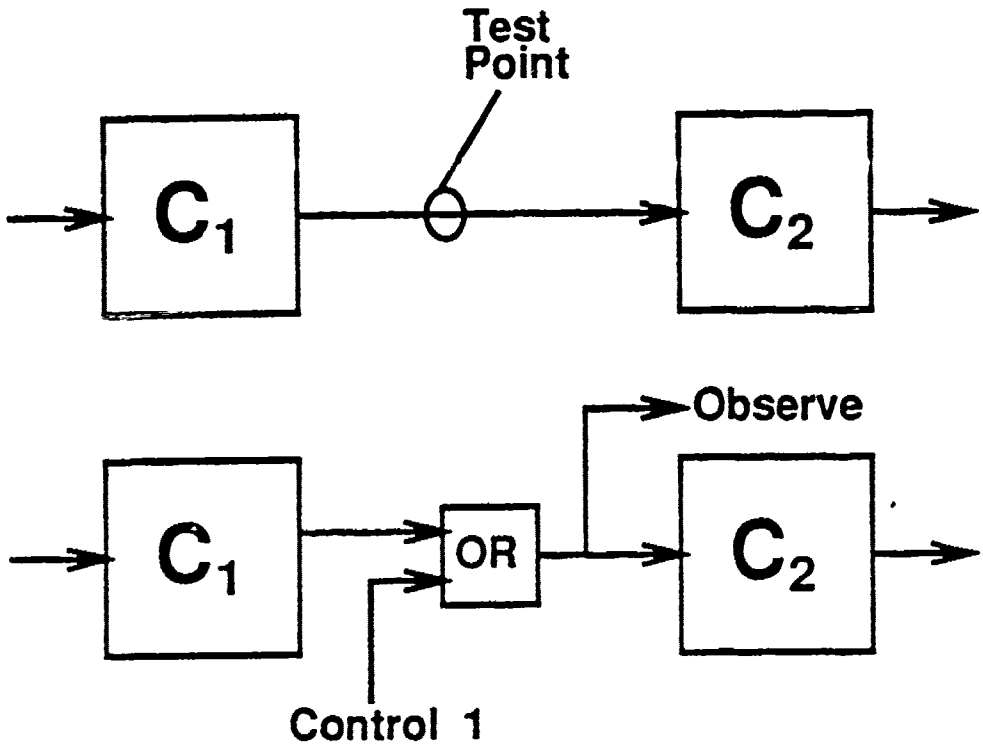


Figure 9 Test point insertion the example shows line control of value 1 and observation

such that the overall gain in controllability and observability is maximum. Since all such measures are approximate, the effect of test points on test generation time and fault coverage can not be accurately predicted. All of the methods rely on a test generator and fault simulation to find which nodes should be the best candidates for test points. One can use a limited trial-and-error method to insert the test points, and then generate a test to accurately see its effect. One can combine the above two methods by first selecting a few test points, based purely on structural analysis, and then augmenting it by more test points selected by a test generator.

When test points are inserted at all flip-flops and only at flip-flops in conjunction with a scan register, the methods are generically called *scan designs*. If the flip-flops are master-slave or edge-triggered then additional flip-flops for the scan register are not needed. The scan register is implemented directly on the original flip-flops in the design. Of course, one still needs additional pins, shift-in, shift-out, shift-clock and load. The load signal may be combined with the system clock, thus saving a pin. A scan design makes all flip-flops completely controllable and observable. This means that the circuit can be placed in any state and its next state completely observed. Since the inputs to the combinational logic either come from primary inputs or from some flip-flops, the scan allows one to apply any test vector to the combinational logic portion of the circuit. And since the outputs from the combinational logic go to primary outputs or to some flip-flops, the responses are completely observable. This method essentially reduces the sequential test problem to a combinational test problem. There are several slightly different scan-based structures used by different computer manufacturers. The most widely known structure is Level Sensitive Scan Design (LSSD), used by IBM in many of its systems. LSSD uses level sensitive (not edge triggered), hazard free latches. Two latches form a master-slave flip-flop. It is estimated that the LSSD scan designs have 10% to 15% area overhead.

Scan has also become a standard for board-level testing. When scan is applied to the periphery of a chip it is termed *boundary scan*. The goal is to make every chip on a board completely controllable and observable from the edge connector of the board. In addition, the boundary scan registers are designed in such a way that they can also be used to test the interconnect between chips. At the board level, predominant failures are in the interconnects and pins. Therefore, boundary scan is very useful in board testing. Of course the chips, themselves, must be designed for testability. Boundary scan only provides access to the chips, not testability on the chips. Boundary

scan is useful only if all chip manufacturers follow a standard method of communication on board. IEEE has put out a standard for boundary scan that many manufacturers have agreed to follow. Further details on a variety of designs for testability techniques can be found in [33, 39].

4.5. Built-In Self-Test (BIST)

.Built-in self-test is the capability of a circuit (chip, board or system) to test itself. Most circuits are tested by external testers, which apply the test vectors and monitor the responses. In BIST circuits, the test vectors are internally generated and applied, and the responses are also internally monitored. A general organization for BIST is shown in Figure 10. The test generator and response monitor have to be very simple to keep the overhead of BIST very small. As a result, the test generator is generally a counter, which produces exhaustive test patterns, or it is a linear feed-back shift register (LFSR) which produces pseudo-random patterns. The response monitor is similarly very simple. The monitor compresses the responses into a single word, called *signature*. Compression methods include, counting number of 1's and 0's in the response stream, counting 0-to-1 or 1-to-0 transitions, forming a checksum, taking parities and so on. The circuits that produce the signature are also called *signature analyzers*. Signature produced by the compression is calculated by simulation of the fault-free machine and stored on the chip. During the actual test of the circuit, the signature is produced by the compressor on chip and then compared with the stored, good signature. A mismatch between signatures indicates a faulty chip. At times, a faulty chip produces the same signature as a good signature and the fault goes undetected. This can happen for two reasons: 1) the test set failed to detect the fault; or, 2) the test set detected the fault but the compressor "lost" the information. The loss of information is always possible in any compressor. For example, a parity compressor will produce the same parity if the faulty responses have an even number of bits in error. This situation is referred to as *error masking*, and the faulty output which produces the same signature as the good output is called an *alias* of the good output. Aliasing probability can be analytically estimated if one can accurately characterize all error responses of a faulty circuit. The actual loss of fault coverage due to aliasing is not so easy to estimate. A fault simulation of the BIST circuit with the signature analyzer in place, can accurately give the loss of coverage.

An Organization for Built-In-Test

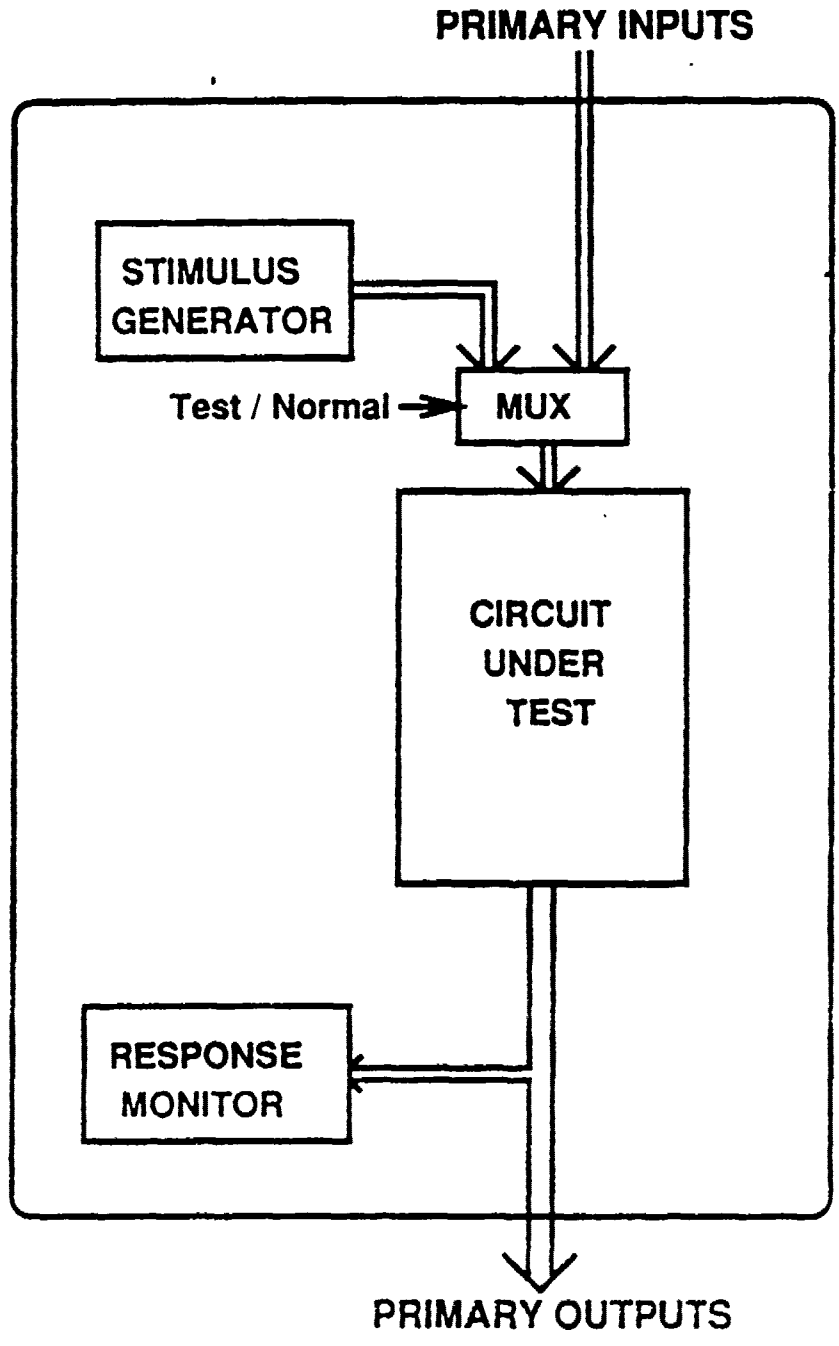


Figure 10 Built-In Self-Test Organization. Stimulus generator usually is a counter or an autonomous LFSR. Response monitor usually is a signature analyzer based on LFSR.

Of all the compression methods, the checksum methods have proven to be the most effective in practice. A checksum is simply an addition of numbers modulo a constant. A modulo addition can also be viewed as a division by a constant in which the quotient is discarded and the remainder is retained. Since in the test context, the numbers themselves have no specific meaning (*i.e.*, they are simply some binary strings) one can use any number system and any constants, as long as the compressor has low hardware complexity and low aliasing. Linear feedback shift registers (LFSRs) can form the checksums in polynomial algebras over the binary strings, and they are easy to design. Aliasing probabilities of LFSRs have been analyzed for specific error models and all analyses lead to the conclusion that aliasing probability is $\frac{1}{2^n}$ for an n-bit LFSR. Experimental studies of some circuits using fault simulators have shown that the loss in fault coverage due to aliasing is usually much less than 1%. More details on pseudo-random techniques for BIST can be found in [40].

5. Evaluation

The foregoing sections have outlined a wide range of techniques useful for designing fault-tolerant systems. In any given situation, the relative efficiency of these techniques must be evaluated so that design trade-offs can be made. Such analysis is an integral part of the design process. The next two sections introduce the question of evaluation and discuss the different methods available to model, analyze and measure the dependability of fault-tolerant systems. In section 5.1, methods to develop analytical models for computer system reliability, availability and performability are outlined. A wide range of automated tools that allow an informed user to conduct evaluations of complex structures have been developed. The characteristics of some of these tools are given. Measurement-based methods for evaluation techniques are discussed in section 5.2.

5.1. Analytical Models

In this section we briefly review computer system dependability modeling issues. We first discuss two widely used combinatorial models [41]. Then we address Markov models, including availability, reliability, and performability (reward) models. Finally, we take a look at six representative software modeling tools.

5.1.1. Simple Models for Fault-Tolerant Systems

If T is a random variable that denotes the lifetime or time-to-failure of a system component (and t its particular value) then T has a cumulative distribution function (CDF) given by

$$F(t) = P(T \leq t) \tag{1}$$

The reliability R(t), of the component is the probability that the component survives until time t:

$$R(t) = P(X > t) = 1 - F(t) \tag{2}$$

Typically, R(t) is assumed to be an exponential distribution thus $R(t) = \exp(-(\lambda)t)$, where λ is failure rate. As explained in section 2, the elementary reliability models of fault-tolerant computing systems are often variations on the so-called *NMR Model* (N-Modular Redundant). The system is composed of n identical and independent components, m or more of which must be functioning for the system to be operational. Thus, the system has (n – m) "hot standby" components. Under these simplifying assumptions, we can express the reliability of the system as:

$$R_{NMR}(t) = \sum_{i=m}^{n} \binom{n}{i} R(t)^i (1-R(t))^{n-i} \tag{3}$$

Special cases include the serial system (m = n), the parallel system (m = 1), and the triple modular redundant voting system (n = 3, m = 2).

The second elementary reliability model represents an N-modular Standby Redundant system (NSR). It has (n – 1) of n identical components maintained in a powered-off, (cold standby) state. Upon failure of the single active component, one of the (n – 1) powered-off components is switched into operation. It is assumed that there is no chance of a failure associated with switching. The system lifetime random variable in this case is the sum of n identical component lifetime random variables, so

$$R_{NSR}(t) = 1 - \int_0^t dF^{(n)} \tag{4}$$

where $dF^{(n)}$ denotes n-fold convolution. If the probability of failure during switching is taken into account, the above expression must be appropriately modified [43].

5.1.2. Markov Models

Markov models allow us to describe complex interactions among components and are widely used. A discrete-state Markov process is a collection of states together with the transition rates among these states. When a Markov process is used to model dependability of a computer system, each state in the model represents a distinct combination of operational and failed states of individual components or modules of the system. The process of failing and recovering of the components is described by the transition from one state to another in the Markov model [42]. In a discrete time Markov process transitions can only occur at fixed intervals, while in a continuous time Markov model transitions can occur at any time. A Markov process has the property that the future state of the process depends only on its present state and not on the past (i.e., it is memoryless). A continuous time system is said to have the Markov property if:

$$P\{X(t+s) = j \mid X(s) = i, X(u) = k, 0 \le u < s\} = P\{X(t+s) = j \mid X(s) = i\} \quad (1)$$

where $s, t \ge 0$ and i, j, k denote the states of the model.

If, in addition,

$$P\{X(t+s) = j \mid X(s) = i\} = P\{X(t) = j \mid X(0) = i\} = p_{ij}, \quad (2)$$

the Markov process is said to be stationary or homogeneous. In other words, a continuous-time, homogeneous Markov model represents a time-evolving process that changes states according to the following rules:

(1) The holding time in each state i is exponentially distributed with mean h_i.

(2) Given that the system is in state i, it goes to state j with a probability (transition probability) p_{ij}.

If the exponential distribution (1) above is not satisfied, (i.e., the distribution is of a general form) the model is said to be semi-Markov. Usually, analytical models assume that the holding time in each state is exponentially distributed. From a practical point of view, this assumption can limit the accuracy of the model results.

The details of the theory and applications of Markov models to reliable systems are given in [43]. Figure 11 shows a Markov model for a simple system with two components. The system is assumed to fail if both components fail (a 1-out-of-2 system). The failures of the two components are assumed to be independent. There are four states in the model: the normal state S_N — both components are operational; the single component failure

state S_i (i = 1, 2) — where component i has failed; the system failure state S_F — where both components have failed. The λ_i and μ_i denote the mean failure rate and recovery rate for component i, respectively.

5.1.2.1. Availability Evaluation

Given that there are n states (1,2,...,n) in a Markov (or semi-Markov) model, then, at any time $t \geq 0$, the state distribution can be expressed by the probability vector

$$P(t) = (p_1(t), p_2(t), ..., p_n(t)) \tag{3}$$

where $p_i(t)$ is the probability that the process is in state i at time t and satisfies the following condition:

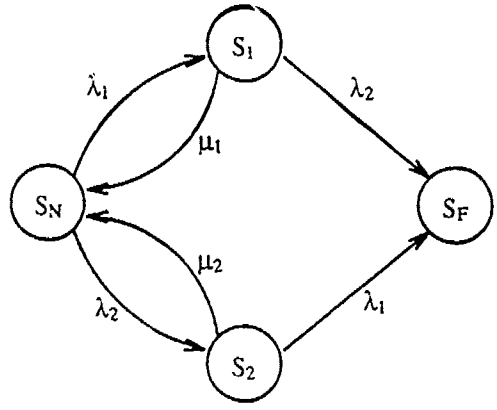
$$\sum_{i=1}^{n} p_i(t) = 1 . \tag{4}$$

Further, assume that the components in the system can be in one of two states: operational or failed. The system is considered operational, or available, if at least a minimum set of components is operational. The failure and repair process of these components can be represented by a Markov (or semi-Markov) model with state space Ψ [44, 45]. We can partition the total set of states (Ψ) into an operational set (Ψ_o) and a failed set (Ψ_f). The probability, A(t), that the system is operational at a time t, is referred to as the *instantaneous availability*:

$$A(t) = \sum_{i \in \Psi_o} p_i(t) \tag{5}$$

The *interval availability*, $\bar{A}(t)$, is the proportion within a given interval of time that the system is operational. This is given by:

Figure 11. Markov Model for a 1-out-of-2 System.



$$\bar{A}(t) = \frac{1}{t}\int_0^t A(x)dx \tag{6}$$

The *steady-state availability*, A, is the limit of the interval availability as t goes to infinity:

$$A = \lim_{t\to\infty} \bar{A}(t) \tag{7}$$

Equation (7) above is equivalent to the following commonly used definition of availability:

$$A = \frac{MTTF}{MTTF+MTTR} \tag{8}$$

5.1.2.2. Reliability Evaluation

To evaluate reliability based on a Markov model, we make all failure states (Ψ_f) be absorbing states so that once the system enters Ψ_f, it is destined to stay there. That is, we modify the model by setting all transition probabilities out of a state in Ψ_f to zero. For example, S_F in Figure 11 is an absorbing state. If this modified model is solved, the system reliability can be evaluated as:

$$R(t) = \sum_{i\in\Psi_o} p_i(t) \tag{9}$$

where Ψ_o is the operational set of states in the model. It is seen that R(t) is a special case of the instantaneous availability when failure states are set to absorbing states.

5.1.2.3. Performability (Reward Rate) Evaluation

In evaluating availability and reliability, we assume that a system is fully-operational (in an up state) without any degradation. If a system is allowed to operate in a degraded mode, a combined measure of performance and availability, called *performability* [46], is often used. Typically, performability can be evaluated via reward models by defining a *reward rate* r_i [47] ($0 \le r_i \le 1$) for state i, rather than simply a zero or a one, as in the case of availability and reliability models. Performability measures are generalizations of availability measures, they fall into three basic classes, namely *instantaneous* reward rate at t,

$$Y(t) = \sum_{i\in\Psi} r_i p_i(t)\ , \tag{10}$$

interval reward rate over t,

$$\bar{Y}(t) = \frac{1}{t}\int_0^t Y(x)dx\ . \tag{11}$$

and *steady-state* reward rate,

$$Y = \lim_{t \to \infty} \overline{Y}(t) . \tag{12}$$

5.1.3. Modeling Tools

Various software tools have been created to evaluate dependability for computer systems, using both analytic and simulation techniques. These software tools are sophisticated and require a user with a good degree of expertise in reliability engineering and computer design. A summary of characteristics of six representative tools is listed in Table 1 [41, 48]. All of these tools can be used to evaluate dependability measures for both repairable and nonrepairable systems. Most are based on Markov models.

5.2. Measurement-Based Analysis

Measurement is an essential part of the evaluation process. In the final analysis, evaluation techniques discussed above must be supported by measurements in the field. A study of production systems is valuable not only for accurate evaluation but also for gaining insight into reliability bottlenecks in system design. Measurements are made through the different stages of design, development and manufacturing and provide the basis for gaining

Table 1. Summary of Characteristics of Six Dependability Evaluation Tools

Tool	HARP[49]	METASAN[50]	SAVE[51]	SHARP[52]	SURE[53]	SURF[54]
Models	Nonhomogeneous Markov	Stochastic activity networks	Fault trees Continuous-state Markov	Directed graphs Fault trees Semi-Markov	Semi-Markov	Markov
Solution Techniques	Runge Kutta Simulation	Gaussian elimin. Iterative method Simulation	SOR* Randomization Simulation	SOR* Laplace	Computation of bounds	Laplace
Input	Any failure distr. Fault tree Markov chain	Description of stochastic activity network	Exp. distr. Fault tree Markov chain	Multistage Exp. distr.; Multiple levels of models	Markov chain	Transition matrix
Output	Reliability Availability	Performability	Availability	Reliability Performance	Reliability	Reliability Availability
Operating Systems	UNIX, VMS MS-DOS	UNIX	VM MVS	UNIX VMS	VMS	IBM TSO

* SOR: Successive overrelaxation

understanding and insight into the system and the manufacturing process.

In some instances, measurements are made directly on production systems in the field (i.e., uncontrolled phenomena). In other instances, experiments are devised in the laboratory under controlled conditions. Faults are deliberately introduced, and their impact on system hardware and software is measured. Both approaches have their relative advantages and disadvantages and are used by manufacturers and researchers as a basis for design and evaluation. The lessons learned are useful for developing improved validation techniques and also to develop fault masking and recovery methods to lessen the impact of defects on the user.

More than a dozen years of research effort have measured, analyzed, and modeled over 80 machine-years of data. Issues ranging from the monitoring of computer reliability to the analysis of the measured data to quantify system dependability (reliability and availability) in the field have been addressed. Laboratory techniques involving a wide variety of fault injection techniques ranging from physical fault insertion to simulation have been developed and tested. The measured hardware and software data have been used not only to characterize the system reliability and fault tolerance in the field, but also to jointly characterize the interdependence between reliability and performance. Measurement-based research has revealed the dependence of failure rates on workload, led to the development of improved diagnosis strategies, and has also contributed to the development of accurate modeling and validation techniques. Finally, such measurements are crucial in evaluating the coverage of different fault tolerance and recovery mechanisms in the system.

5.2.1. Field Measurements

From a research point of view, field measurements have provided much valuable information on actual failure characteristics and their distributions. They provide estimates for parameters used in analytical models. Some examples are component failure rates, coverages and the relative frequency of different types of faults. Often, the interactions among hardware, software, and application programs are complex and hence not easily amenable to analysis. Measurements serve as an exploratory tool to understand the effect of faults on these different system components and their interactions.

Specifically, research based on field experiments has resulted in several significant findings. First, results have shown that the commonly used, simple exponential model is representative of only a small fraction of system failures. Second, the failure distributions are best characterized by the Weibull function [4]. Depending on the failure type, the hazard function can be decreasing, increasing or constant. Finally, both hardware and software failures have a tendency to occur in bursts [55]. Even though the cause of the burst is often a single fault, its effect impacts several components leading to multiple errors or failures. Thus, unless error detection and diagnosis techniques substantially improve, the single point failure assumption common to many system design strategies may not be fully justified.

Importantly, the above investigations also showed that the dependability of both hardware and software was significantly affected by the operational environment of the system. Experimental investigations conducted to quantify this phenomenon are discussed in the next section.

5.2.1.1. Workload Impact on Failure Characteristics

Experimental research, based on over a decade of measurements on several generations of IBM, DEC and other mainframes [56, 57] has established the influence of the level and type of operational workloads on system reliability. Measured error and workload data from IBM and DEC systems under different operational environments have shown that, on the average, the failure rate of a system was four to five times as high, under heavy workloads than at low workloads. On a dynamic level, the measurements showed that the risk of a failure at high workloads was 50 to 100 times greater than that at low loads. These results are significant because, even though some (e.g., process control) computers repetitively execute the same program with effectively the same input requests, most have widely-varying workloads as measured by such metrics as processor utilization. Thus, the results brought into question the validity of conventional reliability models, which do not take the operational environment into account and hence added a new dimension to dependability evaluation.

The dependency of reliability on workload is due to several phenomena. The first is referred to as error latency. As failures occur within a system, they must be detected in order to affect the statistics. Many failures lie dormant (or latent) until a particular module or subsystem is exercised. These latent faults are more likely to

manifest during the high workload conditions since an increase in the workload implies an increase in the state transitions and path executions in the computer. Thus, even if failures are not caused by increased utilization, they are revealed by this factor. Secondly, as system utilization approaches saturation levels, a statistically higher software failure rate results due to increased stress on these programs. Timing and synchronization problems are also more likely to be revealed at high workloads and often these conditions are difficult to reproduce in the laboratory. Also, many load-dependent failures occur in the area of code involved with exception handling. Usually, this section of code is not well debugged. Under high workload conditions, as critical resources get saturated the exception handling code may be executed and reveal software faults and design errors. There is also some evidence to show that higher workloads result in higher operating temperatures and hence in increased failure rates.

The results of these studies are significant. They indicate that it is not useful to push a system close to its performance limits (the generally accepted operational goal). The slight gain in performance improvement is more than offset by the degradation in system reliability. Thus, classical computer reliability models need to be re-evaluated in order to take system workload explicitly into account. This research has had a strong impact on the modeling community. Several researchers, [58, 59] have since proposed analytical models that take workload variations into account. The second impact has been to bring out the importance of validation as an integral part of the modeling process.

5.2.2. Measurement-Based Models

Given the above results, it is reasonable to ask how workload parameters can be taken into account in generating reliability/availability models. One approach is to model the workload as a daily 24 hour cycle and assume a linear relationship between workload and failures. The ensuing model is cyclostationary in nature and has been shown to represent real system behavior [57].

Experimental research has developed methods for identifying and building Markov models of the resource-usage/failure/recovery process directly from measured data. The approach uses sampled system activity parameters to identify headers of usage which can then be identified as a state in a performance/reliability model [60]. At each interval of time the measured workload is represented by a point in four-dimensional space (CPU utilization,

CPU waiting for input/output, I/O controller activity, and disk activity). A statistical cluster analysis technique was used to divide the workload into similar classes. Each cluster was represented as a system state, and a state transition diagram with intercluster transition probabilities was developed.

5.2.2.1. Software Reliability Evaluation

There has been a great deal of research in the area of software reliability evaluation and a large number of models have been proposed. By and large, the term software reliability refers to the manufacture of software. The models are usually empirical in nature and attempt to describe the reliability growth of the candidate software during the manufacturing, debugging and testing phases. A large number of models have been developed. In general the models can be divided into two classes. The first class is based on the number of remaining defects in the software. The simplest such model referred to as the Jelinski-Moranda model [61] assumes that the time to failure is proportional to the number of remaining defects. Also, perfect repair of a software bug is assumed. There are a number of generalizations of this approach. Imperfect debugging, uncertainty in the projected number of initial defects, have all been modeled [62]. The vast majority of these models have been shown to be valid in their measured environments. The second class of models [63] does not depend on knowledge of the number of remaining defects or their distribution. Thus, while most models assume that the failure rate is a function of the number of remaining defects, the Littlewood-Verall model assumes the failure rate is a random variable with a gamma distribution. Thus the software reliability becomes a doubly stochastic process. The concept of the failure rate as a random variable is expected to treat the uncertainty in the efficiency of the bug-fixing process. A comparison of many of the existing models has been made by several researchers [62, 64] using different data sets. Although most models perform well within their own contexts, their performance varies significantly from one data set to another. Thus, no single model can be expected to perform well under all circumstances. In other words, the question of deciding *a priori* as to what is the best model for a given situation remains open at this stage. Additionally, few models address the question of operational reliability of software systems. Studies on the impact of the operating environment on software reliability is given in [57, 65].

5.2.3. Controlled Experiments: Fault Injection

Although field data provides a rich source of information, an adequate number of machine years of data are not always available. Fault injection is an important method to mimic the occurrence of errors in a controlled environment that can be instrumented to make the necessary measurements [66, 67]. Several automated tools to allow both physical and simulated faults have been developed both in academia and in industry. Some of the measurements of interest are *latency* [67] and *coverage* [68, 69].

There are numerous theoretical and practical difficulties associated with making measurements. The question of what to measure, and how to measure it, is indeed a difficult one. From a statistical point of view, sound evaluations require a considerable body of data. The usual assumptions regarding uniform populations and stationarity may not fully hold in computing environments. Fault-injection experiments have known input error distributions but the question remains as to how representative of naturally-occurring errors are those that are selected for injection. The success of such experiments depends on the choice of fault models, a realistic workload, and finally, valid experimental design.

6. Commercial Fault-Tolerant Computing

Fault-tolerant computing has evolved from specialized military and communications systems to general-purpose, high-availability commercial systems. The evolution of fault-tolerant computers has been well documented [4, 76]. The earliest high availability systems were developed in the 1950's by IBM, Univac, and Remington Rand for military applications. In the 1960's, NASA, IBM, SRI, the C. S. Draper Laboratory and the Jet Propulsion laboratory began to apply fault tolerance to the development of guidance computers for aerospace applications. The 1960's also saw the development of the first AT&T electronic switching systems.

The first commercial fault-tolerant machines were introduced by Tandem Computers in the 1970's for use in on-line transaction processing applications [71]. Several other commercial fault-tolerant systems were introduced in the 1980's [72]. Current commercial fault-tolerant systems include distributed memory multi-processors (Tandem NonStop [73], Tolerant Eternity [74]), shared-memory transaction-based systems (Sequoia [75]), 'pair-and-spare' hardware fault-tolerant systems (Stratus [76], DEC VAXft 3000 [75]), and triple-modular-redundant

systems (Tandem Integrity S2).

Most applications of commercial faul -tolerant computers fall into the category of on-line transaction processing. Financial institutions require high availability for electronic funds transfer, control of automatic teller machines, and stock market trading systems. Manufacturers use fault tolerant machines for automated factory control, inventory management, and on-line document access systems. Other applications of fault tolerant machines include reservation systems, government databases, wagering systems, and telecommunications systems.

Vendors of fault tolerant machines attempt to achieve both increased system availability and continuous processing. Depending on the system architecture, either processes continue to run despite failures or the processes are automatically restarted from a recent checkpoint. Some traditional systems have enough redundancy to reconfigure around failed components, but processes running in the failed modules are lost. Vendors of commercial fault-tolerant systems have extended fault tolerance beyond the processors and disks. To make large improvements in reliability, all sources of failure must be addressed, including power supplies, fans and inter-module connections.

The Tandem NonStop and Integrity architecture. will be described to illustrate two current approaches to commercial fault-tolerant computing.

6.1. Tandem NonStop Systems

Tandem NonStop systems are designed to continue operation despite the failure of any single hardware component. In normal operation, each system uses its major components independently and concurrently, rather than as "hot standbys." Figure 11 shows the architecture of the NonStop Cyclone system. A system consists of up to 16 processors interconnected by dual busses. Each processor has its own memory which contains a copy of the message-based Guardian operating system. Each processor controls one or more I/O busses. Dual-porting of I/O controllers and devices provides multiple paths to each device. Disks may be mirrored to maintain redundant permanent data storage

---

NonStop, Guardian, Integrity S2, NonStop Cyclone and NonStop V+ are trademarks of Tandem Computers, Incorporated

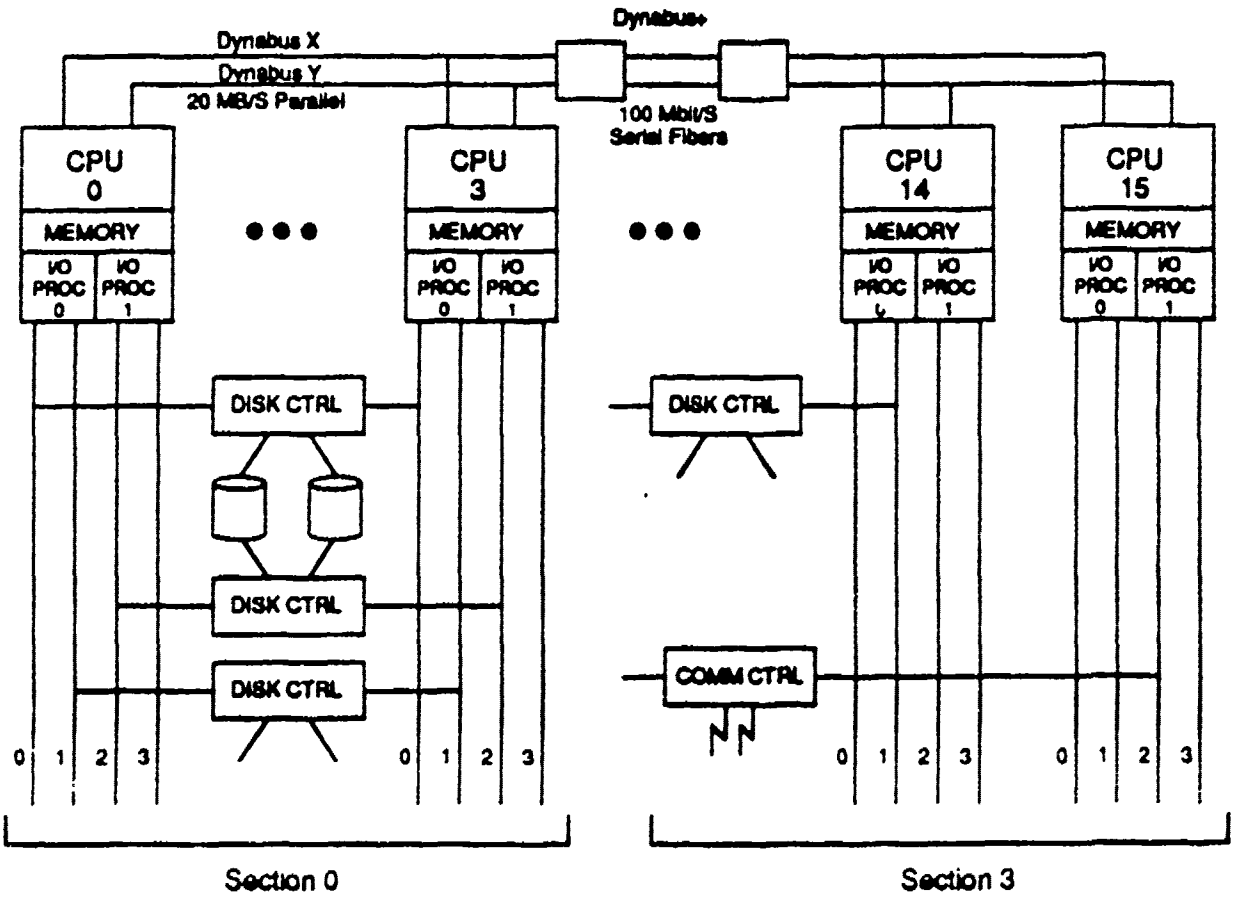


FIGURE 12. NonStop Cyclone system architecture.

Each module has self-checking hardware to provide "fail-fast" operation -- either a module operates correctly, or it stops to prevent contamination of other modules. Faults are detected by parity checking, duplication and comparison, and error detection codes. Fault detection is primarily the responsibility of the hardware, while fault recovery is the responsibility of the software.

Processes under Guardian may run as process-pairs. A primary process runs in one processor and a backup process runs in a different processor. The backup is usually dormant, but periodically updates its state in response to checkpoint messages from the primary. A checkpoint can take the form of a complete state update, or as a delta checkpoint which communicates only the changes from the previous checkpoint. Originally, checkpoints were manually inserted in application programs, but currently most application code runs under transaction processing software which provides recovery through a combination of checkpoints and transaction two-phase commit protocols.

When a processor fails, the failing processor is identified by the absence of periodic "I'm Alive" messages. Guardian directs the appropriate backup processes to begin primary execution from the last checkpoint. New backup processes may be started in another processor, or the process may be run with no backup until the hardware has been repaired.

Each I/O controller is managed by one of the two processors to which it is attached. Management of the controller is periodically switched between the processors. If the managing processor fails, ownership of the controller is automatically switched to the other processor. If the controller fails, access to the data is maintained through another controller.

In addition to providing hardware fault tolerance, process pairs provide some measure of software fault tolerance. When a processor fails due to a software bug, the backup processes frequently are able to continue processing without encountering the same bug. The software environment in the backup processor typically has different queue lengths, table sizes, and process mixes. Since most of the software bugs escaping the software quality assurance tests involve infrequent data dependent boundary conditions, the backup processes often succeed.

Continuous operation requires the capability for faulty modules to be identified, serviced, and reintegrated while the system is on-line. A fault-tolerant diagnostic system monitors system operation, isolates the most likely

failing module, and optionally dials a remote center to request service. Modules such as processor boards, controllers, disks, fans, and power supplies may be replaced on-line.

6.2. Integrity S2

The Integrity S2 illustrates another approach to fault-tolerant computing. S2, which was introduced in 1990, was designed to run a standard version of the UNIX operating system. In systems where compatibility is a major goal, hardware fault recovery is the logical choice since few modifications to the software are required.

A diagram of the Integrity S2 system is shown in Figure 12. The processors and local memories are configured using triple-modular-redundancy (TMR). All processors run the same code stream, but clocking of each module is independent to tolerate faults in the clocking circuits. Execution of the three streams is asynchronous, and may drift several clock periods apart. The streams are re-synchronized periodically and during access of global memory. Voters on the TMR Controller boards detect and mask failures in a processor module.

Memory is partitioned between the local memory on the triplicated processor boards and the global memory on the duplicated TMRC boards. The duplicated portions of the system use self-checking techniques to detect failures. Each global memory is dual ported and is interfaced to the processors as well as to the I/O Processors (IOPs). Each IOP controls a NonStop V+ bus. Standard VME peripheral controllers are interfaced to a pair of NonStop V+ busses through a Bus Interface Module (BIM). If an IOP fails, the BIM switches control of all controllers to the remaining IOP. Mirrored disks may be attached to two different VME controllers.

In Integrity S2, all hardware failures are masked by the redundant hardware. After repair, components are reintegrated on-line.

The preceding examples have shown ways in which commercial vendors have incorporated fault tolerance into data processing systems. Approaches involving software recovery require less redundant hardware, and offer the potential for some software fault tolerance. Hardware approaches use extra hardware redundancy to allow full compatibility with standard operating systems and to transparently run applications which have been developed on other systems. Commercial fault-tolerant computing will grow in importance as companies grow increasingly

[2]UNIX is a trademark of AT&T

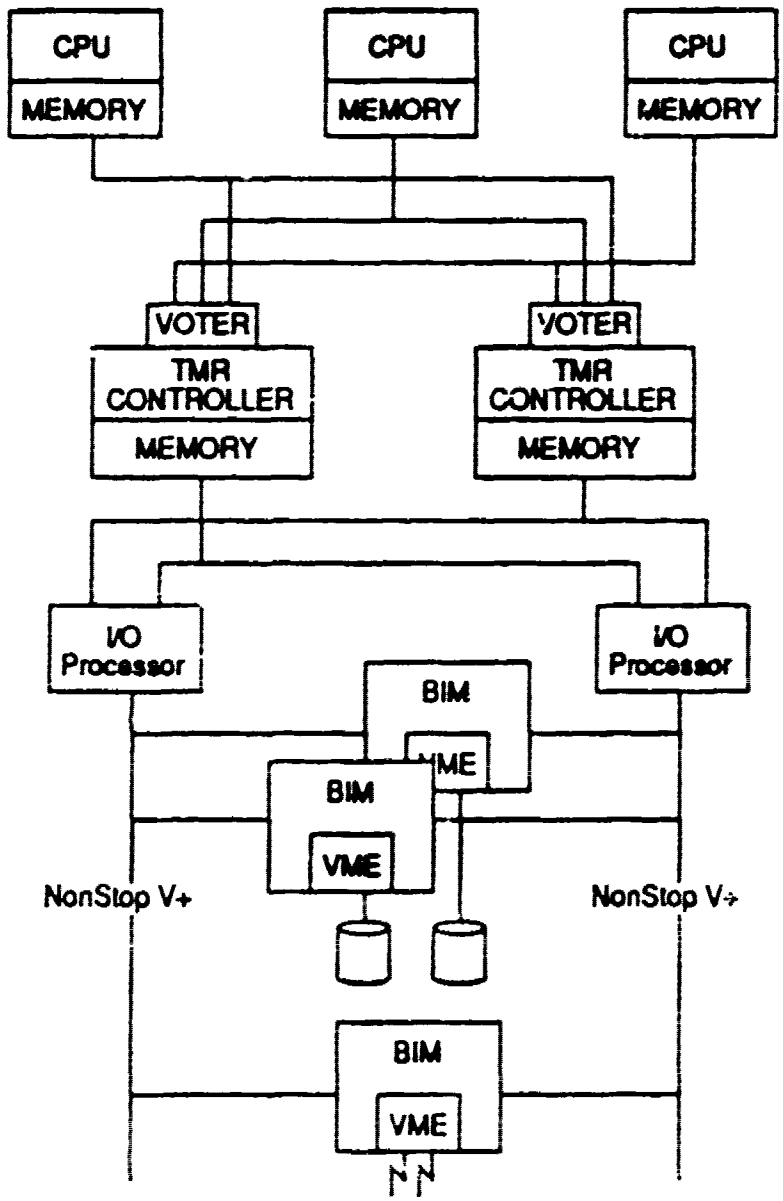


FIGURE 13. Integrity S2 system architecture.

dependent on the correct operation of their computer systems.

7. Acknowledgement

The authors thank Jayne Chase Loseke and Dong Tang for their invaluable assistance in preparing this manuscript. This work was supported by NASA grant NAG-1-613 at the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), a NASA Center of Excellence, the Semiconductor Research Corporation under grant 90-DP-109, and the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under contract N00014-90-J-1270.

References

[1] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies, Annals of Mathematical Studies*, No. 34, (Princeton University Press: Princeton, NJ) pp. 43-98, 1956.

[2] Algirdas Avizienis and Jean-Claude Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proceedings of the IEEE*, Vol. 74, No. 5, May 1986.

[3] D.A. Rennels, "Fault Tolerant Computing--Concepts and Examples," *IEEE Trans. Computers*, Vol. C-33, No. , pp. 1116-1129, Dec. 1984.

[4] B Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, (Addison Wesley: Reading, MA) 1 .

[5] D.P Siewiorek and R.S. Swarz, *Theory and Practice of Reliable System Design*, (Digital Press: Bedford, MA) 1982.

[6] T.R.N. Rao and E. Fujiwara, *Error Control Coding for Computer Systems*, (Prentice-Hall: Englewood Cliffs, NJ) 1989.

[7] R. Blahut, *Theory and Practice of Error Control Codes*, (Addison Wesley: Reading, MA) 1984.

[8] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, (Elsevier North Holland, Inc.: New York, NY) 1978.

[9] D.A. Reynolds and G. Metze, "Fault Detecting Capabilities of Alternating Logic," *IEEE Trans. Computers*, Vol. C-27, No. 12, pp. 1093-1098, Dec. 1978.

[10] J.H. Patel and L.Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Trans. Computers*, Vol. C-31, No. 7, pp. 589-595, July 1982.

[11] K.H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans Computers*, Vol. C-33, No. 6, pp. 518-528, June 1984.

[12] J.A. Abraham, P. Banerjee, C.-Y. Chen, W.K. Fuchs, S.-Y. Kuo, and A.L.N. Reddy, "Fault Tolerance Techniques for Systolic Arrays," *IEEE Computer Magazine (Special Issue on Systolic Arrays From Concept to Implementation)*, Vol. 20, No. 7, pp. 65-77, July 1987.

[13] P. Banerjee, J.T. Rahmeh, C. Stunkel, V.S.S. Nair, K. Roy and J.A. Abraham, "An Evalution of System-Level Fault Tolerance on the Intel Hypercube Multiprocessor," *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pp. 326-367, June 1988.

[14] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans Software Eng.*, Vol SE-1, No 2, pp. 220-232, June 1975.

[15] A. Avizienis, "Software Fault Tolerance," G.X. Ritter, Ed. *Information Processing 89: Proc. IFIP XI World Computer Congress*, (North-Holland: Amsterdam) pp. 491-498, Sept. 1989.

[16] V. P. Nelson, and B. D. Carroll, ed., *Tutorial. Fault-Tolerant Computing*, (IEEE Computer Society. Los Angeles) 1987.

[17] R. J. Abott, "Resourceful Systems for Fault Tolerance, Reliability, and Safety," *ACM Computing Surveys*, Vol. 22, no. 1, pp. 35-68, March 1990.

[18] T. Anderson and P A. Lee, *Fault Tolerance Principles and Practice*. (Prentice Hall: London) 1981.

[19] H. Hecht and M. Hecht, "Fault-Tolerant Software," in *Fault-Tolerant Computing*. D. K. Pradhan, ed., (Prentice Hall: Englewood Cliffs, NJ) pp. 658-695. 1986.

[20] T. Anderson, ed., *Resilient Computing Systems*, Vol. 1, (John Wiley and Sons: New York) 1985.

[21] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," *Proc. IEEE 5th Symp. on Reliability and Distributed Software and Database Systems*. pp. 3-12, Jan. 1986.

[22] N. G. Leveson, "Software Safety: Why, What and How," *ACM Computing Surveys*, Vol. 18, no. 2, pp. 125-163, June 1986.

[23] D. J. Taylor, D. E. Morgan, J. P. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," *IEEE Trans. on Software Engineering*, Vol. SE-6, no. 6, pp. 585-594, Nov. 1980.

[24] C. C. Li, P. P. Chen, W. K. Fuchs "Local Concurrent Error Detection and Correction in Data Structures Using Virtual Backpointers," *IEEE Trans. on Computers*, Vol. 38, no. 11, pp. 1481-1492, Nov. 1989.

[25] C. C. Li, W. K. Fuchs, "CATCH- Compiler Assisted Techniques for Checkpointing," *Proc. 20th Int. Symp. Fault-Tolerant Computing*, pp. 74-81, June 1990.

[26] J. A. Stankovic, ed. *Reliable Distributed System Software*, (IEEE Computer Society: Los Angeles) 1985.

[27] L. Svobodova, "Resilient Distributed Computing," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 3, pp. 257-268, May 1984.

[28] P. J. Denning, "Fault-Tolerant Operating Systems," *ACM Computing Surveys*, Vol. 8, no. 4, pp. 359-389, Dec. 1976.

[29] H. Garcia-Molina, "Reliability Issues for Fully Replicated Distributed Databases," *Computer*, Vol. 15, No. 9, pp. 34-42, Sept. 1982.

[30] K. L. Wu, W. K. Fuchs, J. H. Patel, "Error Recovery in Shared Memory Multiprocessors Using Private Caches," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, no. 2, pp. 231-240, April 1990.

[31] K. L. Wu, W. K. Fuchs, "Recoverable Distributed Shared Virtual Memory, *IEEE Trans. on Computers*, Vol. 39, no. 4, pp. 460-469, April 1990.

[32] K.P. Parker, *Integrating Design and Test: Using CAE Tools for ATE Programming*, (Computer Society Press: Washington, DC) 1987.

[33] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital Systems Testing and Testable Design*, (Computer Science Press: New York, NY) 1990.

[34] D.K. Pradhan, Ed., *Fault-Tolerant Computing Theory and Techniques, Vols. I and II*, (Prentice-Hall: Englewood Cliffs, NJ) 1986.

[35] R.G. Bennetts, *Design of Testable Logic Circuits*, (Addison-Wesley: Reading, MA) 1984.

[36] A. Miczo, *Digital Logic Testing and Simulation*, (Harper & Row: New York) 1986.

[37] H. Fujiwara, *Logic Testing and Design for Testability*, (MIT Press: Cambridge, MA) 1985.

[38] F.F. Tsui, *LSI-VLSI Testability Design*, (McGraw-Hill: New York) 1986.

[39] E. J. McCluskey, *Logic Design Principles, With Emphasis on Testable Semicustom Circuits*, (Prentice-Hall. Englewood Cliffs, NJ) 1986.

[40] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, (Wiley: Somerset, NJ) 1987.

[41] R. Geist and K. Trivedi, "Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques, *IEEE Computer*, Vol. 23, No.7, pp. 52-61, July 1990.

[42] J.C. Laprie, "On Reliability Prediction of Repairable Redundant Digital Structures," *IEEE Trans. on Reliability*, Vol. R-25, pp. 256-258, October 1976.

[43] K.S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, (Prentice-Hall: Englewood Cliffs, NJ) 1982.

[44] J. Arlat and J.C. Laprie, "On the Dependability Evaluation of High Safety Systems," *Proc. 15th Int. Symp. Fault-Tolerant Computing*, pp. 318-323, June 1985.

[45] A. Goyal, S S. Lavenberg, and K.S. Trivedi, "Probabilistic Modeling of Computer System Availability, *Annals of Operations Research*, Vol. 8, pp. 285-306, 1987.

[46] J.F. Meyer, 'Closed-Form Solutions of Performability," *IEEE Trans. on Computers*, Vol. C-31, No. 7, pp. 648-657, July 1982.

[47] A. Reibman, R. Smith, and K. Trivedi, "Markov and Markov Reward Model Transient Analysis: An Overview of Numerical Approaches," *European Journal of Operational Research*, Vol. 40, pp. 257-267, 1989.

[48] A.M. Johnson, Jr. and M. Malek, "Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability," *ACM Computing Surveys*, Vol. 20, No. 4, pp. 228-269, Dec. 1988.

[49] S.J. Bavuso, et. al., "Analysis of Typical Fault-Tolerant Architectures using HARP," *IEEE Trans. on Reliability*, Vol. R-36, No. 2, pp. 176-185, June 1987.

[50] W.H. Sanders and J.F. Meyer, "METASAN: A Performability Evaluation Tool Based on Stochastic Activity Networks," *IEEE Fall Joint Computer Conference*, Dallas, Texas, pp. 807-816, Nov. 1986.

[51] A. Goyal, W.C. Carter, E. de Souza e Silva, S.S. Lavenberg and K.S. Trivedi, "The System Availability Estimator," *Proc. 16th Int. Symp. Fault-Tolerant Computing*, pp. 84-89, July 1986.

[52] R.A. Sahner and K.S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Trans. on Reliability*, Vol. R-36, No. 2, pp. 186-193, June 1987.

[53] R.W. Butler, "An Abstract Language for Specifying Markov Reliability Models," *IEEE Trans. on Reliability*, Vol. R-35, No. 5, pp. 595-601, Dec. 1986.

[54] A. Costes, J.E. Doucet, C. Landrault, J.-C. Laprie, "SURF: A Program for Dependability Evaluation of Complex Fault-Tolerant Computing Systems," *Proc. 11th Int. Symp. Fault-Tolerant Computing*, Portland, ME, pp. 72-78, June 1981.

[55] R.K. Iyer, D.J. Rossetti, and M.C. Hsueh, "Measurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Trans. on Computer Systems*, Vol. 4, No. 3, pp. 214-237, August 1986.

[56] R.K. Iyer, S.E. Butner and E.J. McCluskey, "A Statistical Failure/Load Relationship: Results of a Multicomputer Study," *IEEE Trans. on Computers*, Vol. C-31, No. 7, pp. 697-706, July 1982.

[57] X. Castillo and D.P. Siewiorek, "A Workload Dependent Software Reliability Prediction Model," in *Proc 12th Int. Symp. Fault-Tolerant Computing*, pp. 279-286, June 1982.

[58] J.F. Meyer and L. Wei, "Analysis of Workload Influence on Dependability," *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pp. 84-89, June 1988.

[59] Jurgen Dunkel, "On the Modeling of Workload Dependent Memory Faults," *Proc. 20th Int. Symp Fault-Tolerant Computing*, Newcastle-upon-Tyne, England, pp. 348-355, June 1990.

[60] M.C. Hsueh, R.K. Iyer and K.S. Trivedi, "Performability Modeling Based on Real Data: A Case Study," *IEEE Trans. on Computers*, Vol. 37, No. 4, pp. 478-484, April, 1988.

[61] Z. Jelinski and P. Moranda, "Software Reliability Research," *Statistical Computer Performance Evaluation*, edited by Freiberger, W., (Academic Press: New York) pp. 465-484, 1972.

[62] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, (McGraw-Hill Book Company) 1987.

[63] B. Littlewood, "Theories of Software Reliability: How Good Are They and How Can They Be Improved?" *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 5, pp. 489-500, September 1980.

[64] A.A. Abdel-Ghaly, P.Y. Chan, and B. Littlewood, "Evaluation of Computing Software Reliability Predictions," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 9, pp. 950-967, Sept. 1986.

[65] Iyer, R. K. and Rossetti, D. J., "Effect of System Workload on Operating System Reliability: A Study on IBM 3081," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 12, pp. 1438-1448, Dec. 1985.

[66] G.S. Choi, R.K. Iyer and V.A. Carreno, "Simulated Fault Injection: A Methodology to Evaluate Fault-Tolerant Microprocessor Architectures," *IEEE Trans. Reliability, Special Issue on Experimental Evaluation*, October 1990.

[67] E.W. Czeck and D.P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," *Proc 20th Int. Symp. Fault-Tolerant Computing*, pp. 236-243, June 1990.

[68] R. Chillarege and R.K. Iyer, "Measurement-Based Analysis of Error Latency," *IEEE Trans. on Computers*, Vol. 36, No. 5, pp. 529-537, May 1987.

[69] K.G. Shin and Y.H. Lee, "Measurement and Application of Fault Latency," *IEEE Trans. on Computers*, Vol. C-35, No. 4, pp. 370-375, 1986.

[70] A. Avizienis, H. Kopetz, J. C. Laprie, eds., *The Evolution of Fault Tolerant Computing*, (Springer-Verlag: Vienna) 1987.

[71] J. Bartlett, "A NonStop Kernal," in *Proc. 8th Symp. on Operating System Principles*, pp. 22-29, Dec. 1981.

[72] O. Serlin, "Fault-Tolerant Systems in Commercial Applications," *Computer*, Vol. 17, No. 8, pp. 19-30, August 1984.

[73] R. W. Horst, R. Harris, R. Jardine, "Multiple Instruction Issue in the NonStop Cyclone Processor," in *Proc. 17th Symp. on Computer Architecture*, May 1990.

[74] "How Technology is Cutting Fault-Tolerance Costs," *Electronics*, January 13 1986.

[75] D. P. Siewiorek, "Fault Tolerance in Commercial Computers," *Computer*, Vol. 23, no. 7, pp. 26-37, July 1990.

[76] R. Freiburghouse, "Making Processing Fail-Safe," *Mini-Micro Systems*, pp. 255-264, May 1982.